КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра системного анализа и информационных технологий

А.А. АНДРИАНОВА, Е.Е. ЛАВРЕНТЬЕВА, Р.Г. РУБЦОВА

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ДЛЯ ПРОГРАММИСТОВ ПО КУРСУ «ДОКУМЕНТОВЕДЕНИЕ»

Учебно-методическое пособие

Казань – 2016

УДК 004.43
ББК 32.973.26 – 018.1

Принято на заседании кафедры системного анализа
и информационных технологий
Протокол № 5 от 2 февраля 2016 года

Принято на заседании учебно-методической комиссии Института
Вычислительной математики и информационных технологий
Протокол № 6 от 18 февраля 2016 года

Рецензенты:
кандидат физико-математических наук, доцент кафедры
анализа данных и исследования операций КФУ **В.В.Бандеров;**
кандидат физико-математических наук,
доцент кафедры вычислительной математики КФУ **О.В. Панкратова**

Андрианова А.А., Лаврентьева Е.Е., Рубцова Р.Г.
Лабораторный практикум для программистов по курсу «Документоведение»: Учебно-методическое пособие / А.А. Андрианова, Е.Е. Лаврентьева, Р.Г. Рубцова. – Казань: Казан. ун-т, 2016. – 62 с.

В учебном пособии разбираются основные фазы и этапы создания информационной системы документооборота – определение информационной модели документа, способа хранения и представления данных документа, формирование пользовательского интерфейса систем документооборота, экспорт и импорт информации документа, аспекты применения средств защиты информации в системах документооборота. Пособие создано в поддержку проведения лабораторного практикума по курсу «Документоведение» для студентов направления «Информационная безопасность», но может быть интересна для всех студентов, которые занимаются программированием и разработкой сложных информационных систем.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Документоведение как дисциплина находится на стыке целого ряда разнообразных направлений подготовки специалистов, включая направления, связанные с информационными технологиями. Умение создавать на основе современных информационно-коммуникационных технологий удобные информационные системы поиска, создания, управления документами (или документооборотом предприятия) является важной компетенцией современного ИТ-специалиста. Именно формированию данной компетенции служит дисциплина «Документоведение» в рамках учебного плана по направлению «Информационная безопасность», реализуемая в Казанском (Приволжском) федеральной университете.

Цель практикума по курсу «Документоведение» – формирование у студентов навыков работы с документами в информационных системах и приложениях. В рамках практикума рассмотрены вопросы структурированного хранения документов (базы данных, xml-файлы), организации пользовательского интерфейса для доступа к ним, принципы создания печатных форм документов и внедрения средств защиты информации для документов (цифровая подпись, аутентификация пользователей и пр.).

Организационная форма лабораторного практикума предполагает выполнение индивидуального для каждого студента сквозного проекта. В качестве технологии разработки выбирается язык программирования C# и технология .NET – одна и успешных современных универсальных технологий программирования, удобной для студентов младших курсов, изучающих ИТ. Однако с методологической точки зрения возможно использование и других языков программирования для создания проекта.

Сквозной проект заключается в создании небольшой подсистемы работы с документами для некоторой предметной области (магазина, туристической фирмы, банка, страховой компании и пр.). Обязательным требованием является наличие в подсистеме нескольких типов (минимум двух) взаимодействующих документов, например, счет и документ по его оплате.

Проект разделен на несколько этапов (лабораторных работ), каждый из которых оценивается отдельно. Сдача каждого этапа лабораторной работы предполагает демонстрацию работы программы преподавателю и оформление отчета. Отчет оформляется в произвольном виде, но должен содержать несколько обязательных элементов: краткое описание сделанной работы, код программы с комментариями, несколько основных скриншотов.

Приведем краткую характеристику каждого из этапа проекта.

Этап 1. Описание предметной области и создание модели хранимых данных. Определение модели данных для документа (системы документов). Создание модели данных в виде набора данных ADO.NET.

Этап 2. Организация пользовательского интерфейса для заполнения и редактирования данных документов.

Этап 3. Организация создания печатных форм документов (в виде документов MS Office) или сериализация информации документа для передачи, например, по сети.

Этап 4. Организация защиты информации в документах (использование средств криптографии для обеспечения конфиденциальности или целостности документов).

Таким образом, пособие состоит из четырех разделов, в которых описываются основная задача этапа, теоретические основы его выполнения, технологические моменты, касающиеся выполнения задачи, разобранный пример выполнения задачи (инструментальная среда разработки MS Visual Studio 2010), конкретное задание лабораторной работы и советы по его выполнению, а также описание типовых ошибок.

1. ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. СОЗДАНИЕ МОДЕЛИ ХРАНИМЫХ ДАННЫХ

Любая система документооборота представляет собой средство создания, редактирования, хранения и поиска информации о документах. Документ с этой точки зрения понимается как нечто, фиксирующее факт совершения некоторой операции (заключения договора, оплаты товаров, получения услуг, передачи информации и пр.). Поэтому одной из первых задач при проектировании любой информационной системы, работающей с документами, является определение информационной модели каждого из видов документов, чтобы обеспечить экономное и корректное хранение их данных. Первый этап проекта посвящен определению информационной модели документов информационной системы.

1.1. Информационная модель документа

На данном этапе нам следует создать информационную модель документов, которые оформляются в результате какого-либо вида деятельности. Традиционным подходом при разработке информационной модели документа (системы документов) является подход, пришедший из теории баз данных и объектно-ориентированного программирования.

Когда мы фиксируем какую-либо информацию, описывающую то, что происходит в реальной жизни, мы проводим некоторое ***абстрагирование***, т.е. строим модель, выделяя только те свойства, которые являются ***существенными*** для фиксируемого явления. Многие же свойства остаются неучтенными. Согласно теории баз данных каждый фиксируемый факт становится некоторой ***сущностью***, а его характеристики – ***атрибутами*** этой сущности. Для хранения всех объектов сущности используется ***таблица***, в которой столбцы предназначены для хранения атрибутов сущности. Одна строка таблицы определяет значения всех атрибутов одного объекта сущности.

Например, сущность «Студент» может характеризоваться атрибутами «НомерСтуденческогоБилета», «НомерГруппы», «ФИО», «Факультет», «Специальность», «СреднийБаллУспеваемости». В таблице для каждого из атрибутов

должен быть предусмотрен свой столбец. Тогда таблица может иметь следующий вид:

НомерСтуденческогоБилета	Номер-Группы	ФИО	Факультет	Специальность	СреднийБаллУспеваемости
09113	09-107	Иванов Иван Иванович	ИВМ и ИТ	Информационная безопасность	89
и т.д.					

Представленная строка таблицы описывает все характеристики студента Иванова И.И. (объект сущности «Студент»). Отметим, что далеко не все свойства объекта хранит таблица. Например, любой студент имеет дату рождения, росто-весовые характеристики, цвет волос, хобби, пристрастия в литературе, кино и музыке и пр. Однако эти характеристики не являются важными с точки зрения его описания как студента, поэтому информационная модель сущности «Студент» эти свойства не учитывает.

Отметим некоторые важные правила, которым должны удовлетворять таблицы, хранящие информацию о сущностях. Во-первых, значения атрибутов должны быть неделимыми, т.е. представляться простыми типами данных (символьными строками, числами, логическими значениями и пр.). Во-вторых, строки таблицы должны быть отличными друг от друга, в таблице не должно содержаться двух абсолютно одинаковых строк. В теории баз данных это реализуется выделением специального атрибута, однозначно идентифицирующего сущность. Его называют ***первичным ключом***. Так, для сущности «Человек» этим идентифицирующим атрибутом является номер паспорта (не может быть двух человек с одинаковым номером паспорта). Возможна ситуация, когда объект идентифицируется комбинацией нескольких атрибутов. В этом случае говорят о том, что ключ является ***составным***.

Теперь обратимся к понятию документа. ***Документ*** с точки зрения информационных систем предприятий является средством учета некоторого факта деятельности предприятия. Таким образом, документ определяет некоторую сущность, атрибуты которой и описывают этот свершившийся факт.

Как выделить необходимые атрибуты для хранения документа? Обратимся к ситуациям из жизни. Часто для оформления того или иного документа (заявления, договора, квитанции об оплате и т.д.) нам выдают бланк документа, в который мы должны вписать необходимую информацию. Именно эта информация и является атрибутами сущности данного документа.

При проектировании нового документа нам следует задать вопрос: а какая информация должна быть обязательно зафиксирована в документе, какая информация определяет то, что произошло. Разделив эту информацию на элементарные составляющие, представимые в виде простых типов данных, мы получим атрибуты сущности документа.

Например, спроектируем состав атрибутов сущности документа «Чек в магазине», с которым мы сталкивается практически каждый день. Опишем некоторый факт покупки:

Покупатель Иванов И.И. 30 сентября 2014 года купил в магазине «Эдельвейс» следующие товары: молоко по цене 35 руб, 2 бутылки кефира по цене 42 руб., 4 плитки шоколада по цене 40 руб. Покупатель имел дисконтную карту с номером 83423, предусматривающая скидку 5%. Чек о покупке был проведен через кассу №3 (кассир Сидорова С.С.).

Представленное описание позволяет выделить непосредственно данные, которые описывают именно этот документ (они подчеркнуты). Замена этих данных на другие определяет уже другой факт покупки и, соответственно, другой документ чека, фиксирующий этот факт.

Сделаем несколько замечаний. Во-первых, не все атрибуты необходимы, с точки зрения документа имеется некоторая избыточность. Например, мы редко встречаем чеки, в которых указаны ФИО покупателя. Это свойство, если оно задается, будет учтено посредством свойства номера дисконтной карты, так же как и процент скидки. Поэтому достаточно будет определить в модели документа атрибут «НомерДисконтнойКарты». Как же в этом случае можно будет найти

процент скидки или ФИО покупателя, на которого зарегистрирована дисконтная карта? Проблема решается созданием еще одной таблицы для хранения сущности «ДисконтнаяКарта» с атрибутами «НомерДисконтнойКарты», «ФИО_Обладателя», «ПроцентСкидки». Таким образом, имея в чеке номер дисконтной карты, мы всегда можем обратиться к этой таблице и узнать размер скидки по чеку.

Во-вторых, отметим наличие повторяющихся атрибутов (список товаров и их цен). Очевидно, что неизвестно, сколько товаров будет учтено в конкретном чеке, поэтому отдельным вопросом является способ хранения повторяющихся атрибутов документа. Сейчас стандартным подходом в организации хранения документов является деление всех атрибутов документа на ***атрибуты шапки*** (атрибуты, присутствующие в описании документа один раз) и ***атрибуты табличной части документа*** (повторяющиеся атрибуты). Табличная часть представляет собой своеобразную вложенную таблицу в документ.

Выделим атрибуты документа «Чек о продаже». Ключ данного документа является составным и определяется номером чека и его датой (очевидно, что многие магазины начинают отдельную нумерацию чеков для каждого операционного дня). Все ключевые атрибуты являются атрибутами шапки документа. Кроме того, атрибутами *шапки* являются: номер кассы, фамилия кассира, номер дисконтной карты покупателя, название магазина, общая стоимость чека, сумма скидки. Последние два атрибута являются вычислимыми, т.е. они *избыточны* с точки зрения описания документа, но с точки зрения поиска и представления информации частый пересчет этих значений может быть нежелательным, поэтому такие атрибуты можно указывать в модели документа.

Табличная часть документа хранит информацию о купленных товарах данного чека. Поскольку товаров может быть куплено несколько, эти атрибуты являются повторяющимися и сохраняются в виде вложенной таблицы. Реквизитами табличной части являются штрих-код или название купленного товара, его количество, цена, общая стоимость по строке товара (избыточный атрибут, введенный для устранения частого пересчета).

Хранение чеков организуется в виде двух таблиц – одна таблица хранит реквизиты шапки документа, другая – реквизиты табличной части документа. Заметим, что ключевые атрибуты таблицы чека должны входить в таблицу записей чека. Они организуют связь между двумя таблицами. Такое вхождение называется ***внешним ключом.*** По значениям этих атрибутов для каждого конкретного чека можно найти в таблице детальной информации все записи именно этого чека. Кроме того, наличие такой связи позволит запретить ввод данных о покупке товаров в таблицу детальных записей чека для несуществующего чека.

Таким образом, для хранения чека можно организовать две таблицы (ключевые атрибуты подчеркнуты):

Чек *(НомерЧека, ДатаЧека, НомерКассы, Кассир, НомерДисконтнойКарты, Магазин, ОбщаяСтоимостьЧека, ОбщаяСкидка)*

ЗаписьЧека *(НомерЗаписиЧека, НомерЧека, ДатаЧека, ШтрихКодТовара, Цена, Количество, Стоимость).*

Отметим также, что посредством внешних ключей можно организовать связь таблиц «Чек» и «ДисконтнаяКарта» (о которой говорилось ранее). Такая связь, в частности, позволит по номеру дисконтной карты находить размер скидки и запрещать вводить в чеках номера несуществующих дисконтных карт.

1.2. Технология ADO.NET как средство создания информационной модели документов

Рассмотрим средства языка C#, которые можно использовать для организации хранения документов в программах. Конечно, наиболее подходящим для этих целей является использование баз данных. Однако мы не будем разбирать эти вопросы (для этого предусмотрены отдельные дисциплины). Рассмотрим средства имитации работы с базами данных, которые называют отсоединенными наборами данных технологии ***ADO.NET***.

В языке C# для работы с наборами данных предусмотрено пространство имен ***System.Data***, в котором определена целая система классов для создания набора таблиц, их столбцов, строк, организации связи и поиска информации.

Основным классом набора данных является класс ***DataSet*** (набор данных). Он представляет собой имитацию реляционной базы данных, т.е. хранит коллекцию (список) таблиц (объекты класса ***DataTable***) и связей между таблицами (объекты класса ***DataRelation***). Каждая таблица имеет коллекцию столбцов (объектов класса ***DataColumn***) и строк (объектов класса ***DataRow***).

Алгоритм создания набора данных по готовой информационной модели системы документов прост. Сначала создаем пустой набор данных (базу данных). Затем последовательно создаем таблицы набора данных, добавляя их в список Tables объекта набора данных. При создании таблицы достаточно указать только ее имя. Для каждой таблицы требуется создать все ее столбцы (объекты типа DataColumn). При создании столбца указывается его имя и тип данных с точки зрения типов языка C# (например, для целого числа указывается имя типа System.Int32, для символьных строк – System.String и т.д.). Далее созданный объект-столбец добавляется в коллекцию Columns объекта-таблицы.

Для каждой таблицы следует задать первичные ключи – это свойство объекта-таблицы PrimaryKey. Так как первичный ключ может быть составным, это свойство хранит ссылку на массив объектов DataColumn.

Наконец, следует создать связи между таблицами – объекты класса DataRelation. При создании связи указывается ее имя и два массива объектов-столбцов. Первый массив относится к так называемой родительской таблице (для которой этот набор представляет первичный ключ), второй массив – к дочерней таблице (для которой этот набор столбцов является внешним ключом). Далее созданную связь добавляют в список Relations объекта набора данных (объекта типа DataSet).

Таким образом, будет создан объект типа DataSet, представляющий собой модель набора связанных таблиц, которые хранятся в оперативной памяти компьютера.

Приведем пример создания двух таблиц для хранения документа «Чек»:

```
// создание набора данных
DataSet ds = new DataSet();

// создаем таблицу чеков
DataTable chek = new DataTable("Чеки");
// добавляем таблицу в список таблиц набора данных
ds.Tables.Add(chek);

// формируем список столбцов таблицы чеков
// для каждого столбца указывается имя столбца и тип данных

// атрибут номера чека
DataColumn dc = new DataColumn("НомерЧека",
                               Type.GetType("System.Int32"));
ds.Tables["Чеки"].Columns.Add(dc);
// атрибут даты чека
dc = new DataColumn("ДатаЧека", Type.GetType("System.DateTime"));
ds.Tables["Чеки"].Columns.Add(dc);
// атрибут названия магазина
dc = new DataColumn("Магазин", Type.GetType("System.String"));
ds.Tables["Чеки"].Columns.Add(dc);
// атрибут ФИОКассира
dc = new DataColumn("ФИОКассира", Type.GetType("System.String"));
ds.Tables["Чеки"].Columns.Add(dc);
// атрибут общей стоимости чека
dc = new DataColumn("ОбщаяСтоимость",
                               Type.GetType("System.Int32"));
ds.Tables["Чеки"].Columns.Add(dc);

// описание первичного ключа - массива ссылок на столбцы таблицы
// первичным ключом будет комбинация номера и даты чека
DataColumn[] key = new DataColumn[2]
                { ds.Tables["Чеки"].Columns["НомерЧека"],
                  ds.Tables["Чеки"].Columns["ДатаЧека"] };
                 ds.Tables["Чеки"].PrimaryKey = key;

// создаем таблицу записей чеков
// добавляем таблицу в список таблиц набора данных
ds.Tables.Add(new DataTable("ЗаписьЧека"));

// атрибут номера записи чека
dc = new DataColumn("НомерЗаписиЧека",
                               Type.GetType("System.Int32"));
```

```
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут номера чека
dc = new DataColumn("НомерЧека", Type.GetType("System.Int32"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут даты чека
dc = new DataColumn("ДатаЧека", Type.GetType("System.DateTime"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут название товара
dc = new DataColumn("Товар", Type.GetType("System.String"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут цены товара
dc = new DataColumn("ЦенаТовара", Type.GetType("System.Int32"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут количества товара
dc = new DataColumn("Количество", Type.GetType("System.Int32"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);
// атрибут общей стоимости строки
dc = new DataColumn("Стоимость", Type.GetType("System.Int32"));
ds.Tables["ЗаписьЧека"].Columns.Add(dc);

// описание первичного ключа - массива ссылок на столбцы таблицы
// первичным ключом будет номер записи чека
key = new DataColumn[3]
          { ds.Tables["ЗаписьЧека"].Columns["НомерЗаписиЧека"],
            ds.Tables["ЗаписьЧека"].Columns["НомерЧека"],
            ds.Tables["ЗаписьЧека"].Columns["ДатаЧека"] };
ds.Tables["ЗаписьЧека"].PrimaryKey = key;

// создание связи между таблицами
// указывается имя отношения и два массива связанных полей –
// для родительской и дочерней таблиц
DataRelation rel = new DataRelation("СвязьЧека",
new DataColumn[]{ds.Tables["Чеки"].Columns["НомерЧека"],
                  ds.Tables["Чеки"].Columns["ДатаЧека"]},
new DataColumn[]{ds.Tables["ЗаписьЧека"].Columns["НомерЧека"],
                  ds.Tables["ЗаписьЧека"].Columns["ДатаЧека"]});
//добавляем связь в список связей набора данных
ds.Relations.Add(rel);
```

Очевидно, что оперативная память является временной и следует иметь средства сохранения информации о модели для последующего использования. Класс DataSet имеет средства сохранения и чтения информации (модели и самих

данных) из одного из универсальных текстовых форматов представления структурированных данных XML (eXensible Markup Language):

```
// сохранение структуры и данных в xml-файле
// второй параметр задает, что вместе с данными должна быть
//сохранена схема данных, т.е. указание на структуру хранимых данных
ds.WriteXml("chek.xml", XmlWriteMode.WriteSchema);
```

В результате работы данного кода будет создан файл chek.xml, который будет иметь следующее содержимое:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns=""
                xmlns:xs=http://www.w3.org/2001/XMLSchema
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet"
           msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
      <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Чеки">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="НомерЧека" type="xs:int" />
                <xs:element name="ДатаЧека" type="xs:dateTime" />
                <xs:element name="Магазин" type="xs:string"
                                                 minOccurs="0" />
                <xs:element name="ФИОКассира" type="xs:string"
                                                 minOccurs="0" />
                <xs:element name="ОбщаяСтоимость" type="xs:int"
                                                 minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="ЗаписьЧека">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="НомерЗаписиЧека" type="xs:int" />
                <xs:element name="НомерЧека" type="xs:int" />
                <xs:element name="ДатаЧека" type="xs:dateTime" />
                <xs:element name="Товар" type="xs:string"
                                                 minOccurs="0" />
                <xs:element name="ЦенаТовара" type="xs:int"
                                                 minOccurs="0" />
                <xs:element name="Количество" type="xs:int"
                                                 minOccurs="0" />
                <xs:element name="Стоимость" type="xs:int"
                                                 minOccurs="0" />
              </xs:sequence>
```

```
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
      <xs:unique name="Constraint1" msdata:PrimaryKey="true">
        <xs:selector xpath=".//Чеки" />
        <xs:field xpath="НомерЧека" />
        <xs:field xpath="ДатаЧека" />
      </xs:unique>
      <xs:unique name="ЗаписьЧека_Constraint1"
      msdata:ConstraintName="Constraint1" msdata:PrimaryKey="true">
        <xs:selector xpath=".//ЗаписьЧека" />
        <xs:field xpath="НомерЗаписиЧека" />
        <xs:field xpath="НомерЧека" />
        <xs:field xpath="ДатаЧека" />
      </xs:unique>
      <xs:keyref name="СвязьЧека" refer="Constraint1">
        <xs:selector xpath=".//ЗаписьЧека" />
        <xs:field xpath="НомерЧека" />
        <xs:field xpath="ДатаЧека" />
      </xs:keyref>
    </xs:element>
  </xs:schema>
</NewDataSet>
```

В дальнейшем загрузить и использовать этот файл можно будет с помощью следующих команд:

```
// загрузка структуры данных и информации из xml-файла
// в пустой набор данных
DataSet ds = new DataSet();
ds.ReadXml("chek.xml", XmlReadMode.ReadSchema);
```

1.3. Программные средства работы с наборами данных ADO.NET

Мы создали структуры данных для хранения информации о документе, которые отражают информационную модель документа. Теперь следует рассмотреть вопросы формирования и управления самими данными, т.е. данными конкретных сущностей.

Стандартно к задачам управления данными относят добавление, удаление, изменение и поиск данных.

Рассмотрим примеры выполнения всех этих операций на примере документа «Чек».

1. ***Добавление нового чека.*** При создании новой записи таблицы создается пустой объект типа DataRow по структуре требуемой таблицы. Это делается с помощью метода объекта-таблицы NewRow(). Объект-строка таблицы типа DataRow представляет собой ассоциативный массив, к элементам которого можно обратиться по имени столбца таблицы. Пустую строку заполняем значениями атрибутов добавляемого документа. После созданную и заполненную строку добавляем в список строк Rows объекта-таблицы. Например,

```
// ввод информации о чеке
// создание нового чека
DataRow newrow = ds.Tables["Чеки"].NewRow();
// заполнение атрибутов чека
// для определения номера чека можно узнать количество строк
// в таблице чеков
// например, если в таблице 5 чеков, то номер вновь
// созданного чека должен быть 6
newrow["НомерЧека"] = ds.Tables["Чеки"].Rows.Count+1;
// запоминаем номер чека
int nom = (int)newrow["НомерЧека"];
newrow["ДатаЧека"] = DateTime.Today;
// запоминаем дату чека
DateTime date = (DateTime)newrow["ДатаЧека"];
newrow["Магазин"] = "Магазин1";
newrow["ФИОКассира"] = "Кассир1";
// пока не введены данные о записях чека, общая стоимость равна 0
newrow["ОбщаяСтоимость"] = 0;
// записываем созданную запись в таблицу
ds.Tables["Чеки"].Rows.Add(newrow);
```

В результате после записи документа в xml-файле сформируется следующий фрагмент текста:

```
<Чеки>
  <НомерЧека>1</НомерЧека>
  <ДатаЧека>2014-02-02T00:00:00+04:00</ДатаЧека>
  <Магазин>Магазин1</Магазин>
  <ФИОКассира>Кассир1</ФИОКассира>
  <ОбщаяСтоимость>600</ОбщаяСтоимость>
</Чеки>
```

2. ***Просмотр всех строк таблицы.*** Так как строки и столбцы таблицы объединены в стандартные коллекции (списки), алгоритм просмотра всех объектов таблицы заключается в последовательном просмотре всех элементов соответствующих коллекций. Приведем для примера цикл распечатки всех атрибутов шапки чеков из таблицы «Чеки»:

```
// организуем распечатку краткой информации обо всех чеках
// цикл просмотра всех строк из коллекции Rows таблицы «Чеки»
foreach (DataRow dr in ds.Tables["Чеки"].Rows)
{
        string str = "";
        // цикл просмотра всех столбцов
        // из коллекции Columns таблицы «Чеки»
        foreach (DataColumn dc in ds.Tables["Чеки"].Columns)
          // для каждого столбца печатается имя столбца (Caption) и
          // значение в строке по данному имени dr[dc.Caption]
           str += (dc.Caption + ":" + dr[dc.Caption] + ";\t");
        Console.WriteLine(str);
}
```

3. ***Поиск информации.*** Выделим основные задачи поиска информации в таблицах:

- Поиск всей записи (строки таблицы) **по известному первичному ключу**. Для этого используется функция Find() таблицы с указанием значений атрибутов первичного ключа:

  ```
  // поиск чека с номером 1 на сегодняшнюю дату
  DataRow dr = ds.Tables["Чеки"].Rows.Find
                  (new object[] { 1, DateTime.Today});
  ```

  Результатом является объект-строка, с помощью которого уже легко получить любой атрибут строки по его имени.

- Поиск всех дочерних записей по известному внешнему ключу. Для этого используется функция GetChildRows() объекта-строки (типа DataRow). Параметром функции является имя связи (объекта DataRelation), которая определяет отношение «родитель-потомок».

Результатом работы функции является массив всех найденных дочерних строк из дочерней таблицы:

```
// поиск дочерних записей для объекта dr по связи «СвязьЧека»
DataRow [] dcr=dr.GetChildRows("СвязьЧека");
```

- Поиск родительской записи по известному внешнему ключу. Для этого используется функция GetParentRow() объекта-строки (типа DataRow). Параметром функции является имя связи (объекта DataRelation), которая определяет отношение «родитель-потомок». Результатом работы функции является один объект типа DataRow, который является родительским объектом:

```
// поиск родительской записи для объекта dr
// (таблицы «ЗаписьЧека») по связи «СвязьЧека»
DataRow dpr = dr.GetParentRow("СвязьЧека");
```

- Поиск записей, удовлетворяющих некоторому условию. Для этого используется функция Select() объекта-таблицы (типа DataTable). Параметром функции является строка с логическим выражением, определяющим условие, накладываемое на записи. Результатом работы функции является массив объектов типа DataRow, удовлетворяющих условию:

```
// поиск всех чеков, которые пробивались
// кассиром Сидоровой С.С.
DataRow [] drs = ds.Tables["Чеки"].Select
                          ("ФИОКассира='Сидорова С.С.'");
```

4. ***Удаление строки.*** Удаление записи таблицы делается с помощью метода Delete() класса DataRow, т.е. для удаления следует найти нужную строку, сохранив ссылку на нее, и потом по этой ссылке вызвать метод Delete(), например,

```
// поиск чека с номером 1 на сегодняшнюю дату
DataRow dr = ds.Tables["Чеки"].Rows.Find
      (new object[] { 1, DateTime.Today});
dr.Delete();
```

5. ***Изменение атрибутов строки.*** Для изменения какого-либо атрибута некоторой строки достаточно найти ссылку на эту строку и далее обратиться к атрибуту по его имени. Например, при добавлении новой записи в чек требуется корректировать атрибут ОбщаяСтоимость таблицы «Чеки»:

```
// вводим информацию о записях чека
newrow = ds.Tables["ЗаписьЧека"].NewRow();
newrow["НомерЗаписиЧека"] = ds.Tables["ЗаписьЧека"].Rows.Count + 1;
// пусть ключевые значения для вводимого чека хранится
// в переменных nom и date
newrow["НомерЧека"] = nom;
newrow["ДатаЧека"] = date;
newrow["Товар"] = "Товар1";
newrow["ЦенаТовара"] = 100;
newrow["Количество"] = 3;
newrow["Стоимость"] =
          (int)newrow["ЦенаТовара"] * (int)newrow["Количество"];
// запоминаем стоимость
int cost = (int)newrow["Стоимость"];
ds.Tables["ЗаписьЧека"].Rows.Add(newrow);

// для корректировки общей стоимости чека следует изменить
// запись в таблице чека
// организуем поиск чека – родительского объекта
DataRow dpr = newrow.GetParentRow("СвязьЧека");
// корректируем общую стоимость чека на стоимость
// введенной записи (cost)
dpr["ОбщаяСтоимость"] = (int)dpr["ОбщаяСтоимость"] + cost;
```

Лабораторная работа №1.

Требуется для некоторой предметной области выбрать несколько документов (2-3 документа) и создать информационную модель хранения этих документов в виде xml-файла. Для этого требуется создать консольное C#-приложение, которое осуществляет проектирование и начальное заполнение данными xml-файла (достаточно внести 2-3 объекта-документа для каждого вида).

Советы и типовые ошибки

- Как видно из примеров программного кода, можно **именовать таблицы и столбцы** русскими буквами, однако следует подбирать имена таким образом, чтобы они **выглядели как идентификаторы переменных** (единым словом, при необходимости объединения нескольких слов начинать каждое слово с большой буквы или заменить пробелы знаком подчеркивания). Это становится существенным при чтении xml-файлов, а также при использовании имен столбцов в логическом условии, которое указывается в функции Select().

- Внимательно следите за используемыми именами таблиц и столбцов, пытайтесь использовать единые правила именования таблиц и столбцов, так как в программном коде явно указываются все имена и небольшие опечатки приведут к ошибкам времени выполнения (невозможности найти таблицу или столбец с указанным именем). Желательно в отдельном файле или на отдельном листочке иметь всегда информационную модель, с которой работает приложение, чтобы легко найти все используемые в информационной модели имена.

- К типовой ошибке может привести использование пары переменных типа DataRow (например, dr и dr1) для всех таблиц системы. Например, в результате копирования типовых строк программного кода и неоткорректированных имен переменных может возникнуть ошибки обращения к несуществующим столбцам (они существуют в других таблицах) и прочие ошибки.

- При создании столбцов таблиц внимательно следите за используемыми типами данных. Нередко в силу копирования типовых строк кода забывается изменить тип данных. Это может привести к серьезным проблемам на последующих этапах работы с данными.

2.ОРГАНИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА ДЛЯ РАБОТЫ С ДОКУМЕНТАМИ

Информационные системы документооборота предназначены для широкого круга пользователей, которые вовсе не должны быть знакомыми с современными информационными технологиями и представлением информации в памяти компьютера. Поэтому для таких систем важное место занимает разработка пользовательского интерфейса, который понятен пользователю с любой квалификацией в области информационных технологий.

2.1. Графический пользовательский интерфейс

Традиционным современным подходом к разработке пользовательского интерфейса является разработка графического оконного пользовательского интерфейса. Основным принципом такого интерфейса является представление информации с помощью разнообразных элементов управления, которые также являются окнами. С элементами управления могут происходить различные события (нажатие кнопок, выбор элемента из списка и пр.).

Графический пользовательский интерфейс может реализовываться с помощью различных технологий и архитектур приложения. Перечислим только наиболее широко используемые из них:

- ***Однопользовательская архитектура*** приложения предполагает, что за хранение информации и за пользовательский интерфейс управления ею отвечает одно и то же приложение.
- ***Архитектура «клиент-сервер»*** предполагает два приложения (клиент и сервер), которые общаются между собой. Основная цель сервера – обеспечение хранения информации и управление доступом к ней. Цель клиента – обеспечить удобный пользовательский интерфейс и обеспечение связи с сервером для отправки и получения данных. Данная архитектура является многопользовательской, предполагая возможность множества приложений-клиентов, которые работают с одним сервером по единым правилам.

- Очень распространен частный случай клиент-серверной архитектуры – ***web-приложения.*** Здесь в качестве сервера выступает специальный вид сервера – web-сервер, а в качестве клиента – браузер.

Также немало гибридных архитектур, которые совмещают характеристики отдельных видов архитектур. Так, использование xml-файла с данными, как мы предполагаем в нашем проекте, скорее следует отнести к однопользовательской архитектуре, однако нетрудно реализовать и некоторые свойства клиент-серверной архитектуры при условии синхронизации доступа к файлу с данными.

Каждый из универсальных языков программирования имеет собственные средства и технологии формирования пользовательского интерфейса, в основном, реализованные в виде отдельных библиотек классов. Даже в рамках одного языка программирования могут существовать несколько различных технологий работы с графическим пользовательским интерфейсом. Так, например, язык программирования C# имеет следующие технологии:

- Технология Windows.Forms для создания оконных приложений;
- Более новая и «продвинутая» технология WPF;
- Технологии ASP.NET и Silverlight – технологии формирования пользовательского интерфейса в рамках web-приложений.

Все перечисленные технологии базируются на одном и том же языке и одной и той же базовой идее управления пользовательским интерфейсом, которую называют **обработкой событий. Событием** является любое действие пользователя с окном приложения или элементами управления (движения и нажатия кнопок мыши, ввод данных с клавиатуры, перетаскивание окна формы, изменение ее размеров, ввод данных в текстовое поле, нажатие командных кнопок, выбор пунктов меню, выбор элементов из списков и т.д.). Операционная система прослушивает все события и отправляет информацию о них специальным функциям-обработчикам, которые обеспечивают необходимую реакцию программы (в частности, действия по обработке данных).

Таким образом, при разработке оконного интерфейса (какой бы технологией разработки программист не пользовался) перед программистом стоят две за-

дачи – **спроектировать внешний вид необходимых оконных форм** (как набор элементов управления, расположенных на формах) и **создать набор функций-обработчиков событий с формами и их элементами управления.**

Приведем пример разработки простого оконного интерфейса для обработки организации ввода данных для чеков на покупки в магазине, с помощью технологии Windows.Forms среды .NET, т.е. рассмотрим принципы создания приложения, имитирующего работу кассы в магазине.

Кратко опишем пользовательский интерфейс приложения. Главное окно имеет следующий вид (Рис. 1):

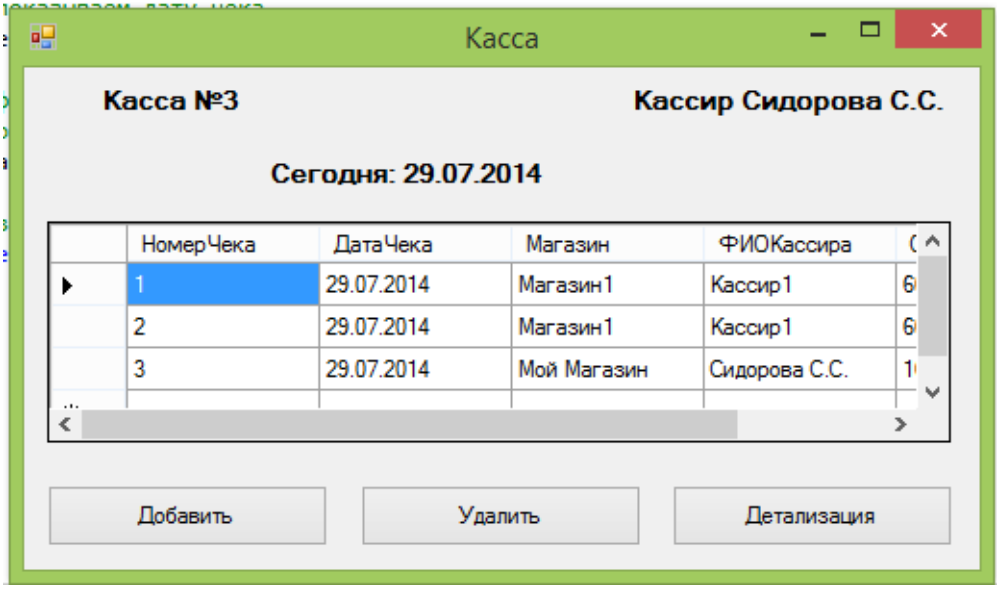

Рис. 1. Вид главного окна приложения управления чеками.

Основу разработки представляет собой элемент управления DataGridView, который представляем собой таблицу с данными уже введенных чеков. Кнопка «Добавить» используется для создания нового чека. В результате ее нажатия помимо создания новой записи чека будет вызываться диалоговое окно (Рис. 2), в котором можно ввести все данные о купленных по данному чеку товарах:

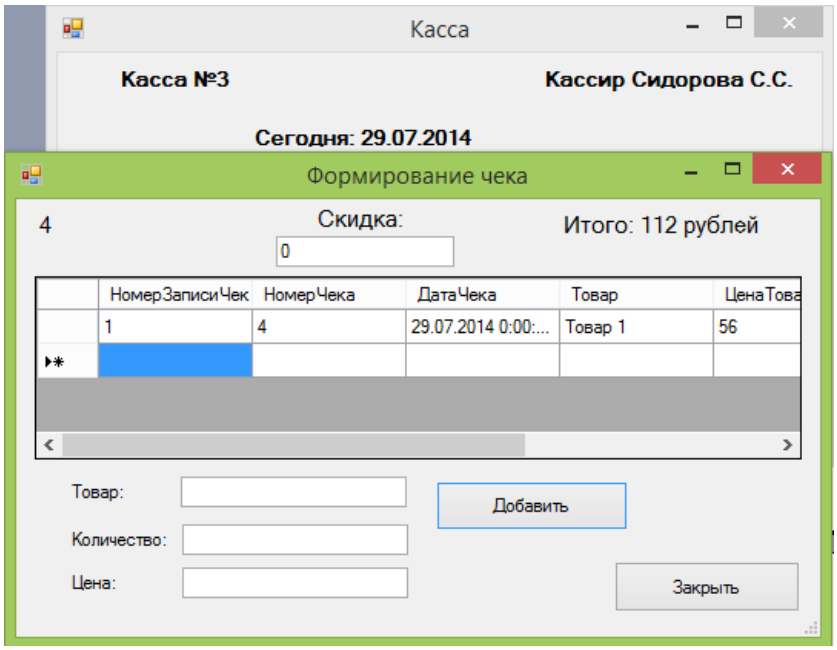

Рис. 2. Окно для формирования нового чека.

Кнопка «Детализация» позволяет вызывать другое диалоговое окно (Рис.3), в котором можно будет только просмотреть данные выбранного чека:

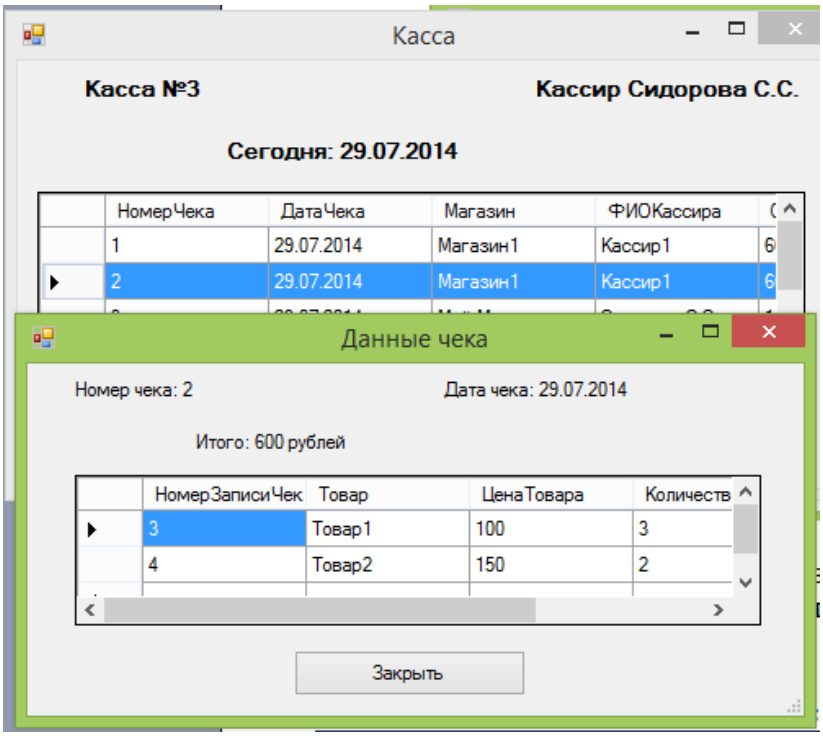

Рис. 3. Окно просмотра детальной информации выбранного чека.

Наконец, нажатие на кнопку «Удалить» должно привести к удалению данных выбранного чека.

2.2. Объектная модель Windows-приложения

Приложение, которое создается по технологии Windows.Forms, имеет следующую архитектуру. Главная функция находится в классе Program, цель которого только инициировать главную форму (окно) приложения. Закрытие этого окна приводит к завершению работы всего приложения. Обычно класс Program создается автоматически в момент создания Windows-приложения и редко подвергается изменениям:

```
static class Program
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            // запуск приложения, в котором функции главного окна
            // выполняет класс Main
            Application.Run(new Main());
        }
    }
```

Любая форма (окно) приложения (главная или нет) определяется классом, который является наследником библиотечного класса Form. Приложение может содержать любое количество классов-форм, каждый из которых определяет отдельное окно приложения. Одно из них является главным. Остальные, как правило, используют в качестве ***диалоговых окон***. Особенность диалогового окна заключается в том, что оно акцентирует внимание пользователя на себе, не давая переключаться на другие окна, с которыми можно будет продолжать работу только после закрытия диалогового окна.

Визуально любая форма состоит из элементов управления. Создаются они на самом деле программным образом, однако для многих IDE (интегрированных сред разработки) характерен визуальный способ формирования внешнего вида форм (установка и расположение на формах элементов управления и задание их

свойств). В IDE Visual Studio для этого используется специальное представление программного кода, которое называют конструктором формы, и специальная панель элементов (Рис. 4), с помощью которой выбранные элементы управления можно перетаскивать на форму. Если «Панель элементов» не доступна, ее можно вызвать с помощью меню «Вид».

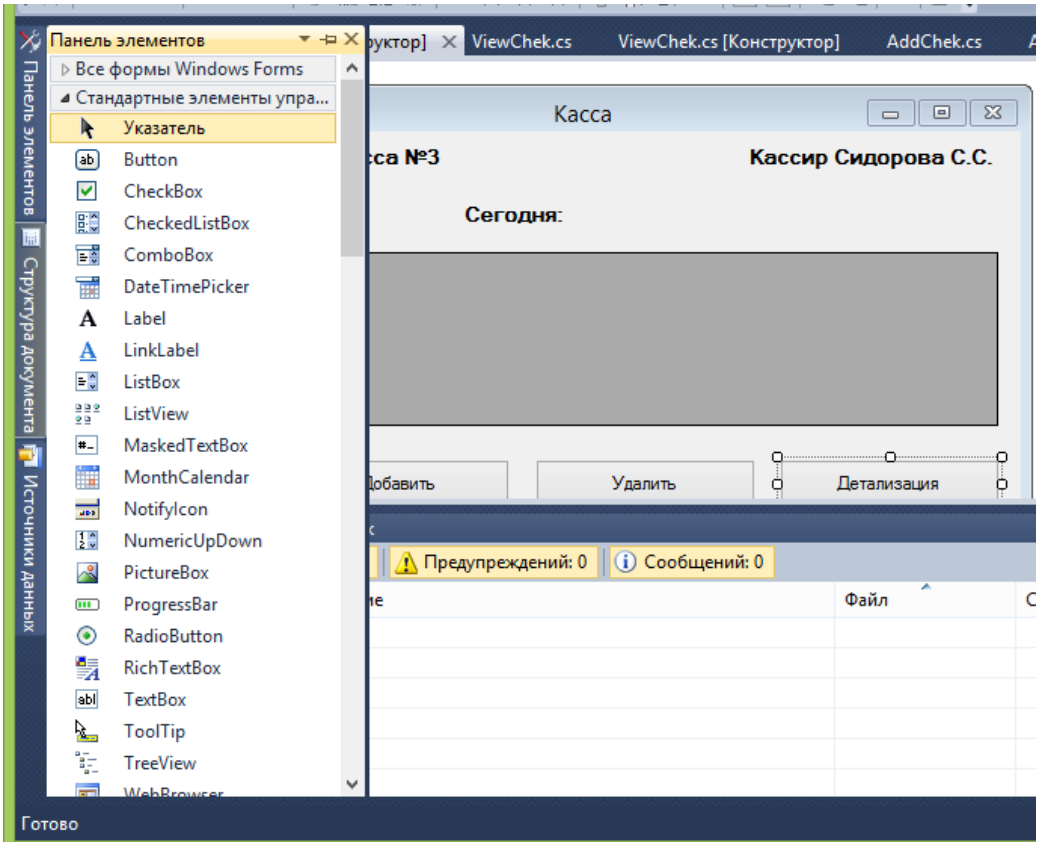

Рис. 4. Конструктор формы и панель инструментов.

Любой элемент управления, помещенный на форму, представляется в классе-форме ***отдельным объектом соответствующего класса (Button, TextBox, Label и т.д.)***, который становится его закрытым (private) свойством. Как и любой объект, они имеют набор свойств и методов, к которым можно будет обращаться программно. Через специальное окно свойств (Рис. 5) можно в режиме конструктора формы редактировать значения свойств окон и элементов управления, например, текст метки (свойство Text), шрифты (свойство Font) и т.д. Отдельное свойство (Name) – это имя объекта-элемента управления, которое можно использовать в программе для обращения к этому объекту (IDE само задаст имя

объектов по умолчанию, однако для программистов эти имена не всегда удобны, например, textBox1 – сложно запомнить, для чего поле предназначалось).

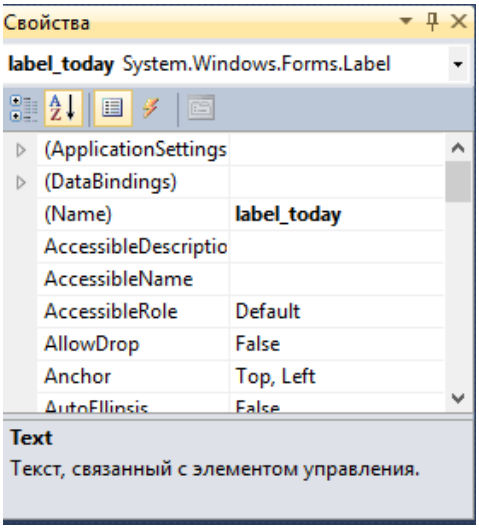

Рис. 5. Окно редактирования свойств элемента управления.

Через это же окно свойств можно создавать обработчики событий конкретных элементов управления. Доступ к списку событий осуществляется с помощью ***кнопки с молнией*** в панели инструментов окна свойств (Рис. 6). Каждый вид элементов управления имеет собственный список событий. Двойной щелчок напротив выбранного события элемента управления в окне свойств приводит к генерации функции-обработчика данного события, которая включается в состав класса-формы.

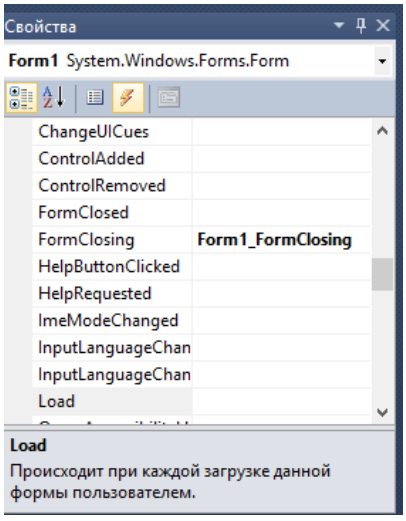

Рис. 6. Окно задания обработчиков событий элементов управления.

Следует отметить, что создание объектов для элементов управления и назначение значений свойств и обработчиков событий, которые устанавливаются визуальным способом, генерируется системой в программный код, который группируется со скрытой части проекта. Обращение к ним осуществляет функция InitializeComponent(), которую следует первой вызывать в конструкторе класса формы.

В нашем примере проект будет включать три класса-формы (Main – класс главного окна приложений, AddChek – класс для формирования состава товаров чека, ViewChek – класс для просмотра данных выбранного чека). Для каждого класса следует определить состав его переменных, конструкторы форм, которые позволят сделать необходимые настройки, и обработчики необходимых событий (в нашем случае это нажатие различных командных кнопок).

2.3. Немного о необходимых элементах управления

Выделим основные задачи, которые потребуется решить при программировании данного пользовательского интерфейса.

1. ***Вызов диалогового окна.*** Создание и вызов диалогового окна производится посредством создания объекта окна диалога с помощью конструктора и вызова метода ShowDialog(), который наследуется от класса Form.

```
// создание объекта класса диалогового окна
// с помощью конструктора
AddChek dlg = new AddChek(nom,date,ds.Tables["ЗаписьЧека"]);
// показ его на экране и передача ему управления
dlg.ShowDialog();
```

2. ***Закрытие диалогового окна.*** Помимо стандартной кнопки закрытия окна, их часто снабжают специальными командными кнопками (OK, Отмена, Закрыть, Да, Нет и т.д), нажатие на которые также должно приводить к закрытию окна. Для того чтобы командную кнопку использовать в этих целях, для нее нужно задать значение свойства DialogResult с помощью окна свойств кнопки (Рис. 7):

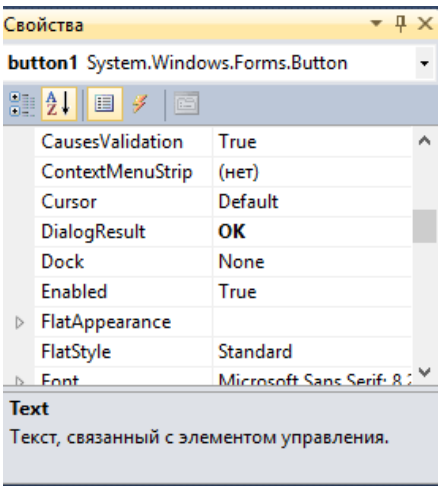

Рис. 7. Задание свойства закрытия окна для кнопки.

3. Установка значений в ***метках и текстовых полях***. Для этих целей предусмотрено свойство Text, например, установка текста метки, в которой отображается операционная дата, осуществляется с помощью следующего программного кода:

```
// формируем строку с датой сегодняшнего дня
label_today.Text = "Сегодня: " +
                    DateTime.Today.ToShortDateString();
```

4. ***Работа с таблицами***. За табличное представление отвечает элемент управления DataGridView. Это очень сложный элемент управления, который имеет множество свойств и событий. Кроме того, в его поддержку создано много вспомогательных классов, отвечающих за отдельный столбец (DataGridViewTextBoxColumn), за отдельную строку (DataGridViewRow) и т.д. Нам необходимо будет решить следующие задачи:
   - Установить для таблицы источник данных – таблицу из набора данных. Это делается с помощью свойства DataSource. Структура столбцов и строк элемента управления DataGridView будет подстроена под источник данных:

```
// установка источника данных для DataGridView для чеков
dataGridView1.DataSource = ds.Tables["Чеки"];
```

- Выбор строк. Список всех выбранных строк таблицы можно получить с помощью свойства SelectedRows. Установить или сбросить выбор конкретной строки можно с помощью свойства класса-строки Selected.
- Получение значения конкретной ячейки таблицы. Любая строка имеет свойство, обозначающее коллекцию ячеек таблицы Cells (ассоциативный массив, к элементам которого можно обращаться по именам столбцов). Значение конкретной ячейки можно получить с помощью применения к ней свойства Value. Следует отметить, что значения в ячейках могут иметь различные типы данных, поэтому обязательно следует применить операцию преобразования в тот тип, которому соответствует получаемое значение. Например, получить номер чека из выбранной строки таблицы чеков можно следующим образом:

```
//получение номера текущего выбранного чека
nom =
  (int)dataGridView1.SelectedRows[0].Cells["НомерЧека"].Value;
```

- В ряде случаев делать настройку источника данных нецелесообразно, например, тогда, когда не нужно показывать все столбцы или все строки источника данных. В этом случае следует создать структуру DataGridView и строки программно. Для каждого столбца создается объект типа DataGridViewTextColumn, настраивается его свойство заголовка (HeaderText), после чего созданный объект добавляется в коллекцию Columns объекта типа DataGridView. Покажем цикл создания структуры таблицы «ЗаписьЧека» без ключевых столбцов родительской таблицы:

```
// формирование DataGridView без автозаполнения
// отмена генерации столбцов DataGridView
dataGridView1.AutoGenerateColumns = false;
// заполнение структуры таблицы
//записи чека для dataGridView1
foreach (DataColumn dc in table.Columns)
```

```
{
    // последовательное создание столбцов элемента управления
    DataGridViewTextBoxColumn dgvc =
                          new DataGridViewTextBoxColumn();
    // заголовок столбца
    dgvc.HeaderText = dc.Caption;
    // не добавляем столбцы с номером и датой чека
    if (dc.Caption.Equals("НомерЧека")
                || dc.Caption.Equals("ДатаЧека"))
          continue;
    // добавление столбца в коллекцию столбцов DataGridView
    dataGridView1.Columns.Add(dgvc);
 }
```

- Программное заполнение строк DataGridView осуществляется посредством пустого объекта типа DataGridViewRow, вызовом для нее функции CreateCells(), которая принимает ссылку на DataGridView для получения структуры строки и всех данных, которые в эту строку должны войти. После созданную строку следует добавить в коллекцию Rows объекта DataGridView. Например, следующим образом осуществляется заполнение данными таблицы товаров заданного чека. Ссылка на строку набора данных с данными выбранного чека содержится в переменной chek:

```
 // заполнение DataGridView данными чека
// находим все дочерние записи для чека
DataRow[] drs = chek.GetChildRows("СвязьЧека");
// заполнение DataGridView данным из полученного массива
foreach (DataRow dr in drs)
{
  // добавляем новую строку в DataGridView с записями чека,
   // заполняем данными из найденной дочерней строки таблицы
  DataGridViewRow dgwr = new DataGridViewRow();
  dgwr.CreateCells(dataGridView1, dr["НомерЗаписиЧека"],
                            dr["Товар"], dr["ЦенаТовара"],
                            dr["Количество"],
                            dr["Стоимость"]);
  // добавляем строку в элемент управления DataGridView
  dataGridView1.Rows.Add(dgwr);
}
```

Далее приведем программный код классов для форм приложения работы с кассовыми чеками.

Класс Main – класс главного окна приложения. Данный класс должен быть дополнен свойством, связанным с источником данных для приложения (xml-файлом, сгенерированным на прошлом этапе сквозного проекта), т.е. в состав класса следует ввести как поле объект класса DataSet:

```
public partial class Main : Form
{
    // источник данных приложения
    DataSet ds = new DataSet();
.     .     .
}
```

Конструктор класса выполняет три цели – задание текущей даты, открытие xml-файла и загрузка его в набор данных, и установка таблицы набора данных в качестве источника данных для DataGridView.

```
public Main()
{
        InitializeComponent();

        // формируем строку с датой сегодняшнего дня
       label_today.Text = "Сегодня: " +
                           DateTime.Today.ToShortDateString();

        // считываем информацию из файла с данными
       ds.ReadXml("chek.xml", XmlReadMode.ReadSchema);

        // установка источника данных для DataGridView для чеков
        dataGridView1.DataSource = ds.Tables["Чеки"];
}
```

Обработчик закрытия формы должен осуществить сохранение набора данных в xml-файл:

```
// обработчик закрытия формы
private void Form1_FormClosing(object sender,
                                          FormClosingEventArgs e)
{
     ds.WriteXml("chek.xml", XmlWriteMode.WriteSchema);
}
```

Обработчик нажатия кнопки «Добавить» должен создавать новую строку таблицы «Чеки» и вызывать диалог задания купленных товаров по чеку:

```
// обработчик нажатия кнопки «Добавить»
private void add_Click(object sender, EventArgs e)
{
        // ввод информации о чеке
        // создание нового чека
        DataRow newrow = ds.Tables["Чеки"].NewRow();
        // заполнение атрибутов чека
        // для определения номера чека можно узнать
        // количество строк в таблице чеков
        newrow["НомерЧека"] = ds.Tables["Чеки"].Rows.Count + 1;
        // запоминаем номер чека
        int nom = (int)newrow["НомерЧека"];
        newrow["ДатаЧека"] = DateTime.Today;
        // запоминаем дату чека
        DateTime date = (DateTime)newrow["ДатаЧека"];
        newrow["Магазин"] = "Мой Магазин";
        newrow["ФИОКассира"] = "Сидорова С.С.";
        // пока не введены данные о записях чека,
        // общая стоимость равна 0
        newrow["ОбщаяСтоимость"] = 0;
        // записываем созданную запись в таблицу
        ds.Tables["Чеки"].Rows.Add(newrow);
        //  выбираем новый чек в качестве текущего
        foreach (DataGridViewRow dgvr in dataGridView1.SelectedRows)
            dgvr.Selected = false;
        dataGridView1.Rows[dataGridView1.Rows.Count - 2].Selected =
                                                              true;
        // вызываем диалог формирования записей чека
        AddChek dlg = new AddChek(nom,date,ds.Tables["ЗаписьЧека"]);
        dlg.ShowDialog();
 }
```

Обработчик события нажатия кнопки «Удалить» фиксирует данные выбранного в DataGridView чека и удаляет его данные:

```
// обработчик нажатия кнопки «Удалить»
private void del_Click(object sender, EventArgs e)
{
      // если чек не был выбран, удалять нечего
      if (dataGridView1.SelectedRows.Count == 0)
           return;
```

```
        //получение номера текущего выбранного чека
        int nom =
         (int)dataGridView1.SelectedRows[0].Cells["НомерЧека"].Value;
        //получение даты текущего выбранного чека
        DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                              Cells["ДатаЧека"].Value;
         // поиск чека по ключу
         DataRow dr = ds.Tables["Чеки"].Rows.Find
                        (new object[] { (object)nom, (object)date });
         dr.Delete();
}
```

Обработчик события нажатия кнопки «Детализация» структурно аналогичен предыдущему – фиксируются данные выбранного чека и для него вызывается диалог просмотра детальной информации по чеку:

```
// обработчик нажатия кнопки «Детализация»
private void view_Click(object sender, EventArgs e)
{
        // если чек не был выбран, удалять нечего
        if (dataGridView1.SelectedRows.Count == 0)
            return;
         //получение номера текущего выбранного чека
         int nom = (int)dataGridView1.SelectedRows[0].
                                          Cells["НомерЧека"].Value;
         //получение даты текущего выбранного чека
         DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                            Cells["ДатаЧека"].Value;
         // вызываем диалог показа детализации выбранного чека
         ViewChek dlg = new ViewChek(ds.Tables["ЗаписьЧека"],
             ds.Tables["Чеки"].Rows.Find(new object[] { nom,date}));
         dlg.ShowDialog();
 }
```

Класс AddChek – класс окна добавления информации о купленных товарах. При проектировании полей класса следует учесть, что работа идет с одним конкретным чеком и его ключевые поля должны быть известны, добавление идет только в таблицу «ЗаписиЧеков», поэтому весь набор данных не нужен, и наконец, нужно будет отслеживать количество записей в чеке и общую стоимость.

Для упрощения выполнения задачи, скидку по карте рассматривать не будем, и ее размер будет по умолчанию установлен в 0.

```
public partial class AddChek : Form
{
    int nom;         // номер текущего чека
    DateTime date;  // дата текущего чека
    DataTable table;    // таблица записей чека
    int count=0;      // количество записей чека
    int total_cost = 0;   // общая стоимость чека
    .      .     .
}
```

Конструктор класса должен принять в качестве параметров ключевые данные чека и ссылку на объект таблицы записей чека, в которую в дальнейшем будет осуществляться вставка новых данных. Элемент DataGridView нельзя будет связывать с источником данных напрямую, так как тогда в нем будут отображаться все данные таблицы. Поэтому в конструкторе осуществляется также настройка структуры данных этого элемента управления:

```
// конструктор, которому передаем ключевые поля
// формируемого чека и таблицу дляя введения туда данных
public AddChek(int nom, DateTime date, DataTable table)
{
      InitializeComponent();

      // сохраняем данные чека
      this.nom = nom;
      this.date = date;
      this.table = table;

      // устанавливаем отсутсвие скидки по умолчанию
      textBox_Discont.Text = "0";
      // делаем поле скидки недоступным
      textBox_Discont.Enabled = true;

      // показываем номер чека
      label_nomer.Text = "" + nom;

      // формирование DataGridView без автозаполнения
      // отмена генерации столбцов DataGridView
      dataGridView1.AutoGenerateColumns = false;
```

```
        // заполнение структуры таблицы записи чека для dataGridView1
        foreach (DataColumn dc in table.Columns)
        {
            // последовательное создание столбцов элемента управления
            DataGridViewTextBoxColumn dgvc =
                                new DataGridViewTextBoxColumn();
            // заголовок столбца
            dgvc.HeaderText = dc.Caption;
            // добавление столбца в коллекцию столбцов DataGridView
            dataGridView1.Columns.Add(dgvc);
         }
    }
```

Также, этот класс должен содержать обработчик нажатия кнопки «Добавить», которая осуществляет считывание данных о купленном товаре из текстовых полей, создание новой строки таблицы записей чека и отображение этой строки в DataGridView. Кроме того, осуществляется корректировка поля «ОбщаяСтоимость» в родительской таблице «Чеки»:

```
// обработчик кнопки добавления нового товара к чеку
private void button_add_Click(object sender, EventArgs e)
{
        // вводим информацию о записи чека
        DataRow newrow = table.NewRow();
        newrow["НомерЗаписиЧека"] = count + 1;
        newrow["НомерЧека"] = nom;
        newrow["ДатаЧека"] = date;
        // считываем данные из текстовых полей о товаре,
        // количестве и цене
        newrow["Товар"] = textBox_Tovar.Text;
        newrow["ЦенаТовара"] = int.Parse(textBox_Price.Text);
        newrow["Количество"] = int.Parse(textBox_Count.Text);
        newrow["Стоимость"] = (int)newrow["ЦенаТовара"]
                                    * (int)newrow["Количество"];
        // запоминаем стоимость
        int cost = (int)newrow["Стоимость"];
        // добавляем запись о купленном товаре в таблицу
        table.Rows.Add(newrow);

        // добавляем новую строку в DataGridView с записями чека,
        // заполняем данными из новой строки таблицы
        DataGridViewRow dgwr = new DataGridViewRow();
```

```
            dgwr.CreateCells(dataGridView1, newrow["НомерЗаписиЧека"],
                 newrow["НомерЧека"], newrow["ДатаЧека"], newrow["Товар"],
                 newrow["ЦенаТовара"], newrow["Количество"],
                 newrow["Стоимость"]);
            dataGridView1.Rows.Add(dgwr);

            // очищаем текстовые поля с данными нового покупаемого товара
            textBox_Tovar.Text = "";
            textBox_Price.Text = "";
            textBox_Count.Text = "";

            // корректировка общей стоимости
            total_cost = total_cost + cost;
            label_total.Text = "Итого: " + total_cost + " рублей";

            // корректировка общей стоимости в родительской таблице
            DataRow dr = newrow.GetParentRow("СвязьЧека");
            dr["ОбщаяСтоимость"] = total_cost;

            count++;
        }
```

Класс ViewChek – класс окна просмотра детальной информации о чеке. Этот класс проще, так как ему достаточно иметь только конструктор, который получает в качестве параметров все необходимые для отображения данные и осуществляет настройку элементов управления.

```
        // конструктор
        // параметры - таблица для поиска и запись с атрибутами шапки чека
        public ViewChek(DataTable table, DataRow chek)
        {
            InitializeComponent();

            // показываем номер чека
            label_nomer.Text = "Номер чека: " + chek["НомерЧека"];
            // показываем дату чека
            label_date.Text = "Дата чека: " +
                      ((DateTime)chek["ДатаЧека"]).ToShortDateString();

             // формирование DataGridView без автозаполнения
             // отмена генерации столбцов DataGridView
             dataGridView1.AutoGenerateColumns = false;
```

```
        // заполнение структуры таблицы записи
        // чека для dataGridView1
        foreach (DataColumn dc in table.Columns)
        {
            // последовательное создание столбцов
            // элемента управления dataGridView
            DataGridViewTextBoxColumn dgvc =
                                    new DataGridViewTextBoxColumn();
            // заголовок столбца
            dgvc.HeaderText = dc.Caption;
            // не добавляем столбцы с номером и датой чека
            if (dc.Caption.Equals("НомерЧека") ||
                      dc.Caption.Equals("ДатаЧека"))
                   continue;
             // добавление столбца в коллекцию столбцов DataGridView
             dataGridView1.Columns.Add(dgvc);
        }
        // заполнение DataGridView данными чека
        // находим все дочерние записи для чека
        DataRow[] drs = chek.GetChildRows("СвязьЧека");
        // заполнение DataGridView данным из полученного массива
        foreach (DataRow dr in drs)
        {
             DataGridViewRow dgwr = new DataGridViewRow();
             dgwr.CreateCells(dataGridView1, dr["НомерЗаписиЧека"],
                dr["Товар"], dr["ЦенаТовара"], dr["Количество"],
                dr["Стоимость"]);
             dataGridView1.Rows.Add(dgwr);
        }
        // формирование записи об итоговой стоимости по чеку
        label_total.Text = "Итого: " +
                                    chek["ОбщаяСтоимость"]+" рублей";
    }
```

Лабораторная работа №2.

Для разработанной на этапе 1 информационной модели документов создать Windows-приложение, обеспечивающее удобный пользовательский интерфейс для ввода и редактирования данных документов.

Советы и типовые ошибки

- Очень внимательно следует отнестись к именованию элементов управления, с которыми нужно будет работать. Если, к примеру, текстовых полей много, то стандартное именование textBox1, textBox2, textBox3 и т.д. может очень быстро запутать, так как при использовании имени в программе программист не помнит его роли и цели. Аналогично, может возникнуть путаница с именами обработчиков событий, так как их имена включают в себя имена элементов управления, которые их вызывают.
- Не забывайте назначать обработчики событий с помощью окна свойств элемента управления. Нередко при копировании и корректировке типового кода программист забывает это сделать. В результате, например, на нажатие кнопки никаких действий не происходит, точнее, просто не вызывается предназначенная для этого функция.
- Удаление обработчика некоторого события желательно осуществлять также с помощью окна свойств. При удалении обработчика некоторого события только из программного возникнет синтаксическая ошибка, связанная с тем, что при формировании обработчика была сгенерирована строка связи, которая находится в скрытом программном коде. Придется удалить ее тоже.
- Не забывайте, что текстовые поля вводят информацию только в строковом виде. Поэтому, если поля должны содержать числа (цены, количества и т.д.), то следует использовать функцию Parse() для получения введенного значения.

3. ЭКСПОРТ ИНФОРМАЦИИ ДОКУМЕНТОВ

Вопросы обмена данными информационной системы с другими приложениями являются важными при проектировании любой информационной системы. Обмен данными включает, очевидно, обе фазы экспорта и импорта информации. Для их реализации необходимо использовать структуры и представления данных, которые понятны всем приложениям, участвующим в процессе обмена. Для простоты ограничимся вопросами экспорта, т.е. сохранения информации документов в форматах, доступных другим приложениям.

Мы рассмотрим две из часто возникающих задач экспорта информации в системах электронного документооборота.

Конечно, просмотр данных документов не ограничивается только пользовательским интерфейсом приложения для ввода и редактирования информации. Электронный документооборот не так давно стал общепринятым. Многие документы согласно законодательству должны храниться в бумажной форме (или в электронной форме, понятной каждому пользователю). Очевидно, что наиболее общепринятым форматом на данный момент для этих целей является **формат MS Word.** Поэтому средства генерации doc-файлов в системах электронного документооборота становятся практически обязательной компонентой.

Еще одной обязательной компонентой являются функции импорта и экспорта данных в другие форматы для передачи данных. Современное состояние информационных технологий позволяет создавать приложения в сложной распределенной архитектуре, в которой компоненты сильно распределены в пространстве. Естественно, что для корректного функционирования требуется налаживать обмен данными между его компонентами, а, возможно, и другими приложениями (информационными системами налоговых органов, информационными системам предприятий-партнеров и т.д.). Поэтому так важен вопрос экспорта и импорта данных документов в понятные другим приложениям форматы. Одна из современных технологий обмена информацией связана с понятием ***сериализации***. Под сериализацией можно понимать процесс сохранения в

определенном формате данных таким образом, чтобы из этого формата можно было бы однозначно произвести обратное преобразование (чтение информации, называемое десериализацией). Существует несколько стандартов сериализации, которые часто используются в приложениях. Наиболее часто применяется бинарный формат, в котором данные представляются в виде набора байтов (содержимое памяти, в которой хранится объект), и xml-формат (существует несколько различных форматов xml, наиболее часто используется SOAP-формат).

3.1. Инкапсуляция данных документа

При любом экспорте и импорте информации удобно использовать объектный подход для ***инкапсуляции*** тех данных, которые мы в дальнейшем должны сохранить или получить извне. Использовать непосредственно наборы данных для этого не совсем удобно, так как обычно для экспорта и импорта требуются не все данные набора данных, или отдельной его таблицы, а вполне конкретные данные, удовлетворяющие определенным условиям и предназначенные для конкретной цели обмена. Поэтому гораздо удобнее инкапсулировать только нужные данные в рамках отдельного объекта и затем работать только с ним.

Разберем сразу на примере подход к инкапсуляции документа «Чек». Пусть для обмена данными требуется отправлять информацию об отдельном чеке (для его аннулирования или подтверждения каких-либо покупок и т.д.). Так как информация чеков хранится в нашем приложении с помощью двух таблиц (таблица атрибутов шапки и таблица атрибутов табличной части), следует создать два взаимосвязанных класса для хранения этих двух групп атрибутов. Взаимосвязь классов организуем посредством внедрения в класс, отвечающий за весь документ (шапку документа), списка объектов, отвечающих за хранение записей табличной части документа.

Забегая вперед, скажем, что для целей последующей сериализации объекта классы должны иметь атрибут [Serializable], конструктор по умолчанию и свойства для доступа к полям класса. Приведем код этих классов. Этот код не нуждается в особых комментариях, отметим только, что поле списка в классе чека предназначено для хранения списка объектов-записей чека, а для его заполнения

предусмотрен метод AddZ() (позволяющий добавить новую строку табличной части в объект-документ).

```
// класс для одной записи чека
[Serializable]
class CheckZ
{
     // поля соответствуют информационным атрибутам
     //строки табличной части чека
     string товар;
     int ценаТовара;
     int количество;
     int стоимость;

     // конструктор по умолчанию, нужен для сериализации
     public CheckZ()
     {
       товар = "";
       ценаТовара = 0;
       количество = 0;
       стоимость = 0;
     }

     // конструктор, инициализирующий запись
     public CheckZ(string s, int price, int count, int cost)
    {
       товар = s;
       ценаТовара = price;
       количество = count;
       стоимость = cost;
    }

    // свойства для доступа к полям класса
    public string Товар
    {
        get { return товар; }
        set { товар = value; }
    }

    public int ЦенаТовара
    {
       get { return ценаТовара; }
       set { ценаТовара = value; }
    }
```

```
        public int Количество
        {
            get { return количество; }
            set { количество = value; }
        }

        public int Стоимость
        {
            get { return стоимость; }
            set { стоимость = value; }
        }
    }

    // класс для инкапсуляции информации всего документа «Чек»
    [Serializable]
    class Check
    {
         // поля соответствуют атрибутам шапки документа
         int номерЧека;
         DateTime датаЧека;
         string магазин;
         string фиоКассира;
         int общаяСтоимость;

         // список для хранения строк табличной части документа
         ArrayList list;

         // конструктор по умолчанию, нужен для сериализации
         public Check()
         {
            номерЧека = 0;
            датаЧека = DateTime.Today;
            магазин = "";
            фиоКассира = "";
            общаяСтоимость = 0;
            list = new ArrayList();
         }

         // конструктор, инициализирующий атрибуты шапки чека
         public Check(int nom, DateTime d, string shop, string kassa)
         {
             номерЧека = nom;
             датаЧека = d;
             магазин = shop;
             фиоКассира = kassa;
```

```
        // общую стоимость следует синхронизировать
        //с табличной частью
        общаяСтоимость = 0;
        list = new ArrayList();
    }

    // метод добавления записи к табличной части документа,
    // заполненный объект записи чека метод получает как параметр
    public void AddZ(CheckZ z)
    {
        // добавляем объект записи в список
        list.Add(z);
        // корректировка общей стоимости чека
        общаяСтоимость += z.Стоимость;
    }

// свойства для доступа к полям класса
public int НомерЧека
{
    get { return номерЧека; }
    set { номерЧека = value; }
}
public DateTime ДатаЧека
{
    get { return датаЧека; }
    set { датаЧека = value; }
}
public string Магазин
{
    get { return магазин; }
    set { магазин = value; }
}
public string ФИОКассира
{
    get { return фиоКассира; }
    set { фиоКассира = value; }
}
public int ОбщаяСтоимость
{
    get { return общаяСтоимость; }
    set { общаяСтоимость = value; }
}
public ArrayList Список
{
    get { return list; }
```

```
            set { list = value; }
        }
    }
```

Создадим в проекте (например, в классе главного окна) функцию, которая по заданным ключевым данным чека формирует инкапсулирующие объекты.

```
// метод инкапсуляции данных о чеке
Check CreateCheck(int nom, DateTime date)
{
      // поиск чека по ключу
      DataRow dr = ds.Tables["Чеки"].Rows.Find
                (new object[] { (object)nom, (object)date });
      // создаем объект чека для последующей сериализации
      Check chek = new Check((int)dr["НомерЧека"],
                            (DateTime)dr["ДатаЧека"],
                            (string)dr["Магазин"],
                            (string)dr["ФИОКассира"]);
      // выбираем все записи, соответствующие
      // выбранному чеку - дочерние записи
      DataRow[] drs = dr.GetChildRows("СвязьЧека");
      foreach (DataRow d in drs)
      {
          // формируем объект записи чека и добавляем его
          // в объект чека
          CheckZ z = new CheckZ((string)d["Товар"],
                                (int)d["ЦенаТовара"],
                                (int)d["Количество"],
                                (int)d["Стоимость"]);
           chek.AddZ(z);
      }
      return chek;
}
```

Далее эта функция будет использоваться для различных целей экспорта данных, а также при рассмотрении вопросов защиты информации, которые возникнут на следующем этапе.

3.2. Создание документа MS Word

Для создания из приложения документа MS Word требуется организовать взаимодействие двух приложений. Действительно, документ формата *doc* или *docx* – это не простые текстовые документы. Они содержат большое количество дополнительной информации, определяющей форматирование текста. Естественно, что знать, как формировать такую информацию может только само приложение MS Word. Поэтому и требуется обращение к программному API (программный интерфейс) этого приложения.

Уже достаточно давно для целей вызова некоторых функций одного приложения из другого была разработана ***модель программирования***, которую назвали ***компонентной (COM – Component Object Model).*** В основе этой модели лежит выделение основного API приложения в виде отдельной dll-библиотеки (библиотеки динамического связывания) и возможность подключения этой библиотеки при разработке приложений практически на любом универсальном языке программирования, поддерживающем COM-модель. Такая модель имеется, естественно, и у всех продуктов семейства MS Office, так как именно к этим приложениям в прикладных приложениях обращаются наиболее часто.

Каким образом можно подключить в проект нужную dll-библиотеку? Для этого в C#-проектах предусмотрены элементы под названием ***Ссылки*** (Рис. 8).

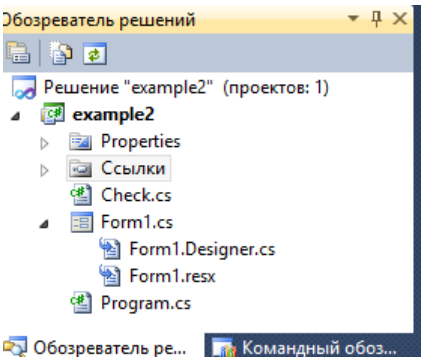

Рис. 8. Окно проекта и пункт «Ссылки».

При добавлении новой ссылки будет открыт диалог (Рис. 9), в котором нужно указать нужную dll-библиотеку.

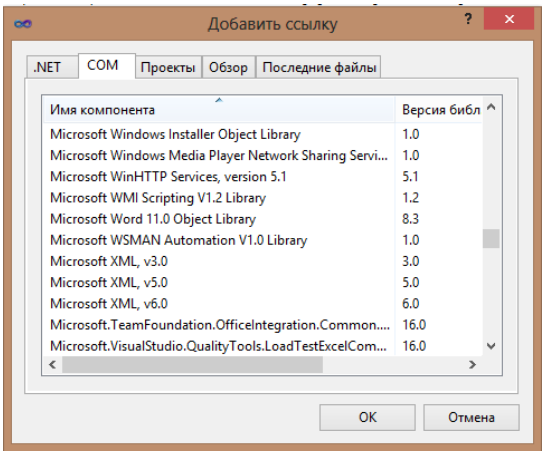

Рис. 9. Окно выбора ссылки на библиотеку для добавления.

Все доступные dll-библиотеки сгруппированы в этом окне по способу регистрации относительно операционной системы. Так, на вкладке .NET приведен список всех dll-библиотек, которые входят в состав .NET Framework, на вкладке COM сгруппированы библиотеки COM-объектов – API различных приложений, которые зарегистрированы в операционной системе, с помощью вкладок ***Проекты*** и ***Обзор*** можно задать путь на сторонние и собственные dll-библиотеки.

Итак, для подключения API MS Word требуется выбрать COM-библиотеку, которая называется «Microsoft Word (номер версии) Object Library» (номер версии здесь зависит от того, какая версия Office установлена на компьютере), поэтому для разных компьютеров она может отличаться.

Что входит в состав этой библиотеки? Библиотека содержит набор классов, представляющей составляющие приложения и документа Word: приложение, документ, абзац, таблица и пр. Для каждой из составляющих определен специальный класс. Классы взаимодействуют друг с другом по принципу «часть-целое». Так, объект-приложение содержит коллекцию документов, объект-документ содержит коллекцию абзацев и пр. Поэтому создание документа MS Word из приложения будет представлять собой последовательное создание объектов от целого к частям и задание их свойств.

Разберем, как программно обращаться к MS Word и генерировать документ, на примере генерации печатной формы чека.

Добавим на главное окно приложения кнопку «Сохранить чек». Для обращения к API MS Word требуется добавить строку подключения пространства имен:

```
using Word=Microsoft.Office.Interop.Word;
```

Имя **Word** далее используется как краткое обозначение этого пространства имен.

Теперь приведем обработчик нажатия кнопки с комментариями:

```
// обработчик нажатия кнопки "Сохранить документ"
private void doc_Click(object sender, EventArgs e)
{
     // определяем, для какого чека следует создать печатную форму
     if (dataGridView1.SelectedRows.Count == 0)
         return;
     // получение номера текущего выбранного чека
     int nom = (int)dataGridView1.SelectedRows[0].
                                    Cells["НомерЧека"].Value;
     // получение даты текущего выбранного чека
     DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                    Cells["ДатаЧека"].Value;
     // инкапсуляция данных для показа в печатной форме
     Check chek = CreateCheck(nom, date);

     // работа с документами Word
     // создание объекта-приложения
     Word.Application app = new Word.Application();
     // создание и добавление объекта-документа MS Word
     Word.Document doc = app.Documents.Add();

     // создание параграфа с заголовком
     // (указание номера и даты чека)
     Word.Paragraph p = doc.Content.Paragraphs.Add();
     // задание текста параграфа
     p.Range.Text = "Чек №" + chek.НомерЧека +
                                    " от " + chek.ДатаЧека;
     // указание, что шрифт должен быть полужирным
     p.Range.Font.Bold = 1;
     // центрирование абзаца
     p.Format.Alignment =
                 Word.WdParagraphAlignment.wdAlignParagraphCenter;
     // устанавливаем межабзацный отступ
```

```
p.Format.SpaceAfter = 20;
p.Range.InsertParagraphAfter();

// вставка параграфа с указанием магазина
p = doc.Content.Paragraphs.Add();
p.Range.Text = "Магазин: " + chek.Магазин;
p.Format.Alignment =
            Word.WdParagraphAlignment.wdAlignParagraphLeft;
p.Format.SpaceAfter = 20;
p.Range.InsertParagraphAfter();

// вставка параграфа с указанием кассира
p = doc.Content.Paragraphs.Add();
p.Range.Text = "Кассир: " + chek.ФИОКассира;
p.Format.Alignment =
            Word.WdParagraphAlignment.wdAlignParagraphLeft;
p.Format.SpaceAfter = 20;
p.Range.InsertParagraphAfter();

// вставка параграфа с указанием общей суммы по чеку
p = doc.Content.Paragraphs.Add();
p.Range.Text = "Сумма: " + chek.ОбщаяСтоимость;
p.Format.Alignment =
            Word.WdParagraphAlignment.wdAlignParagraphLeft;
p.Range.Font.Size = 20;
p.Format.SpaceAfter = 20;
p.Range.InsertParagraphAfter();

// вставка параграфа с таблицей, в которой указана детальная
// информация о купленных товарах
p = doc.Content.Paragraphs.Add();

// при создании таблицы указывается ее
// количество строк и столбцов (2 и 3 параметры)
Word.Table tab =
          doc.Tables.Add(p.Range, 1 + chek.Список.Count, 4);
// указание, что таблица должна иметь рамку
tab.Borders.Enable = 1;

// заполняем ячейки таблицы – обращение к таблице
// осуществляется с помощью функции
// Cell(номер строки, номер столбца)
// отметим, что нумерация строк и столбцов начинается с 1
tab.Cell(1, 1).Range.Text = "Товар";
tab.Cell(1, 2).Range.Text = "Цена";
```

```
        tab.Cell(1, 3).Range.Text = "Количество";
        tab.Cell(1, 4).Range.Text = "Стоимость";

        // просматриваем список купленных товаров и
        // заполняем остальные строки таблицы
        for (int i = 0; i < chek.Список.Count; i++)
        {
            tab.Cell(i + 2, 1).Range.Text =
                        (chek.Список[i] as CheckZ).Товар;
            tab.Cell(i + 2, 2).Range.Text =
                        "" + (chek.Список[i] as CheckZ).ЦенаТовара;
            tab.Cell(i + 2, 3).Range.Text =
                        "" + (chek.Список[i] as CheckZ).Количество;
            tab.Cell(i + 2, 4).Range.Text =
                        "" + (chek.Список[i] as CheckZ).Стоимость;
        }
        // сохранение документа
        doc.Save();
        // активируем окно MS Word для просмотра
        // сгенерированного документа
        app.Visible = true;
}
```

В результате будет получен файл следующего вида (Рис. 10):

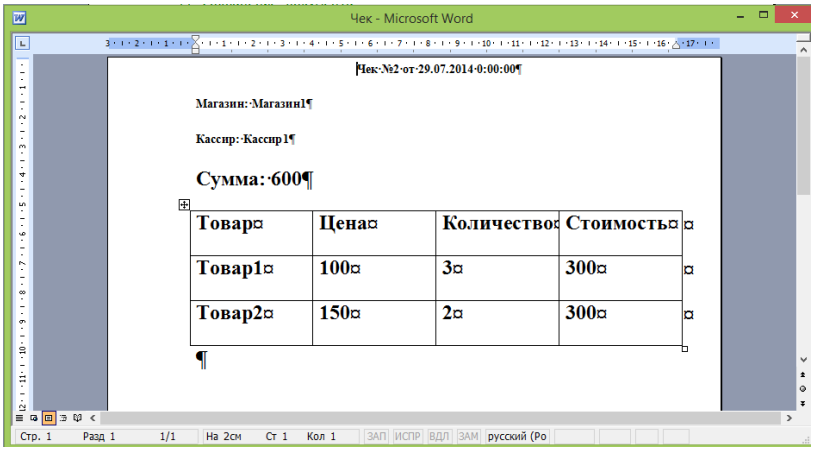

Рис. 10. Окно MS Word с открытым сгенерированным файлом.

Понятно, что чаще всего печатные формы документов имеют гораздо более сложный вид, но это лишь вопрос сложности алгоритма заполнения, а не программных аспектов применения API MS Word.

3.3. Сериализация данных документа

Разберем теперь, как использовать механизм сериализации (преобразование в последовательную форму). Для сериализации обычно используют два вида – бинарную и soap-сериализацию. В общем, ***сериализация*** – это сохранение целостного объекта с возможностью его восстановления (этот процесс называют десериализацией). Сериализация производится в поток, т.е. в файл, в память, в канал связи с другим приложением и т.д. Бинарная сериализация сохраняет двоичный код представления объекта, т.е. полностью содержимое той области памяти, по которой хранится объект. Так как при десериализации это содержимое будет восстановлено, оно также, как и исходный объект, будет однозначно проинтерпретировано.

SOAP-сериализация генерирует xml-представление определенного формата, которое содержит как структуру объекта (из каких полей объект состоит, какие типы у этих объектов), так и все его данные.

Независимо от типа применяемой сериализации, механизм ее использования один и тот же. При сериализации объекта требуется сделать:

- Подключить то пространство имен, которое поставляет классы для выбранного типа сериализации. Так, для бинарной сериализации требуется подключить:

```
using System.Runtime.Serialization.Formatters.Binary;
```

А для SOAP-сериализации:

```
using System.Runtime.Serialization.Formatters.Soap;
```

Отметим, что SOAP-сериализация не входит в подключаемый по умолчанию пакет пространств имен. Поэтому данное пространство имен потребуется подключить с помощью папки ***Ссылки*** в окне проекта (находится на вкладке .NET).

- Создать поток, в который будет происходить сериализация (открыть файл, создать поток в памяти и пр.).
- Создать форматер – специальный объект, который «знает», как нужно сериализовывать объект.
- Вызвать метод форматера Serialize, который и осуществляет сериализацию. Параметрами метода являются поток, куда сериализуется объект, и сам объект, подлежащий сериализации.
- Если поток, в который сериализуется объект, связан с файлом, то для сохранения требуется его закрыть.

В случае десериализации используется функция *Deserialize*, которая получает единственный параметр – поток, откуда десериализуется объект. Эта функция возвращает ссылку на десериализуемый объект.

Приведем теперь программный код сериализации объекта-чека, применительно к нашему учебному приложению. Добавим на форму кнопку «Сериализовать чек», обработчик которой будет создавать файл формата xml, в который будет сериализоваться выбранный пользователем чек. Приведем обработчик нажатия этой кнопки. Также, как и в предыдущем примере, сначала создаем объект **Check** с данными выбранного пользователем чека, который и будет сериализоваться и затем сохранять свой SOAP-формат в выбранный пользователем файл.

```
// обработчик нажатия кнопки "Сериализовать объект"
private void serial_Click(object sender, EventArgs e)
{
      // определяем, для какого чека следует создать печатную форму
      if (dataGridView1.SelectedRows.Count == 0)
           return;
      //получение номера текущего выбранного чека
      int nom = (int)dataGridView1.SelectedRows[0].
                                  Cells["НомерЧека"].Value;
      //получение даты текущего выбранного чека
      DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                  Cells["ДатаЧека"].Value;
      // инкапсуляция данных для сериализации
      Check chek = CreateCheck(nom, date);
      // вызов стандартного диалога сохранения файла
      SaveFileDialog dlg = new SaveFileDialog();
```

```
        if (dlg.ShowDialog() == DialogResult.OK)
        {
             // создание файлового потока, в который
             // будет сериализоваться информация о чеке
             FileStream fs = new FileStream(dlg.FileName,
                                            FileMode.Create);
             // создание сериализатора
             SoapFormatter ser = new SoapFormatter();
             // сериализация объекта chek
             ser.Serialize(fs, chek);
             // закрытие файла
             fs.Close();
             MessageBox.Show("Сформирован файл "+dlg.FileName);
         }
    }
```

Сформированный xml-файл будет выглядеть следующим образом:

```
<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/ "
   xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
       <a1:Check id="ref-1"
        xmlns:a1="http://schemas.microsoft.com/clr/nsassem/example2/
        example2%2C%20Version%3D1.0.0.0%2C%20Culture
        %3Dneutral%2C%20PublicKeyToken%3Dnull">
           <номерЧека>1</номерЧека>
           <датаЧека>2014-02-02T00:00:00.0000000+04:00</датаЧека>
           <магазин id="ref-3">Магазин1</магазин>
           <фиоКассира id="ref-4">Кассир1</фиоКассира>
           <общаяСтоимость>1200</общаяСтоимость>
           <list href="#ref-5"/>
       </a1:Check>
      <a2:ArrayList id="ref-5"
         xmlns:a2="http://schemas.microsoft.com/clr/ns/
         System.Collections">
          <_items href="#ref-6"/>
          <_size>2</_size>
          <_version>2</_version>
      </a2:ArrayList>
      <SOAP-ENC:Array id="ref-6"
                      SOAP-ENC:arrayType="xsd:anyType[4]">
          <item href="#ref-7"/>
          <item href="#ref-8"/>
```

```
          </SOAP-ENC:Array>
      <a1:CheckZ id="ref-7"
           xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
           example2/example2%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral
           %2C%20PublicKeyToken%3Dnull">
         <товар id="ref-9">Товар1</товар>
         <ценаТовара>100</ценаТовара>
         <количество>3</количество>
         <стоимость>300</стоимость>
      </a1:CheckZ>
      <a1:CheckZ id="ref-8"
           xmlns:a1="http://schemas.microsoft.com/clr/nsassem/example2/
           example2%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral
           %2C%20PublicKeyToken%3Dnull">
         <товар id="ref-10">Товар2</товар>
         <ценаТовара>150</ценаТовара>
         <количество>2</количество>
         <стоимость>300</стоимость>
      </a1:CheckZ>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Лабораторная работа №3

Внести в приложение, созданное на этапе 2, возможности сериализации документов и генерации печатных форм в виде документов MS Word.

Советы и типовые ошибки

- Следует быть внимательными при создании инкапсулированного объекта для сериализации, нужно убедиться, что все требования выполнены – атрибут [Serializable], конструктор по умолчанию и обеспечение доступа ко всем полям класса.
- В некоторых случаях при создании документа MS Word рекомендуется для различных параграфов или таблиц использовать различные переменные – не производится обновление объекта, за счет чего может быть неправильное отображение данных.

4. ОРГАНИЗАЦИЯ ЗАЩИТЫ ИНФОРМАЦИИ ДОКУМЕНТОВ

Одними из главных задач при создании систем электронного документооборота являются обеспечение ***конфиденциальности*** информации и поддержание ***целостности*** документов. Обе задачи, в частности, могут решаться применением средств криптографии для обеспечения шифрования информации. Конфиденциальность данных документа достигается посредством того, что лица, имеющие право доступа к данным документа, должны обладать ключами, с помощью которых информация может быть расшифрована. Целостность документов посредством тех же механизмов криптографии позволяет проверить, проводились ли с документом изменения несанкционированным образом или нет.

Разберем, какие средства имеет язык программирования C#, чтобы применить криптографические методы защиты информации. Для этих целей предусмотрено пространство имен ***System.Security.Criptography***, которое содержит классы для поддержки всех основных криптографических механизмов, среди которых наиболее популярны:

- Хэширование (MD5, SHA1 и пр.);
- симметричное шифрование (DES, RS2, Triple-DES, Rijndaen);
- асимметричное шифрование (RSA, DSA).

Вкратце разберем эти механизмы. Каждый из них реализован с помощью специального ***класса-провайдера***, который с помощью своих методов умеет шифровать информацию и в некоторых случаях расшифровывать.

Хэширование предполагает шифрование «в одну сторону» без расшифровки. Его часто применяют тогда, когда требуется сохранять секретные данные пользователя, которые, к примеру, используются для аутентификации пользователя. Например, в базе данных сохраняются данные о паролях клиентов. Хранение явных представлений паролей в базе данных небезопасно. Поэтому перед сохранением пароль хэшируется и сохраняется в базе данных хэш-представление пароля пользователя. Алгоритм хэширования известен только приложению регистрации пользователя (или приложению изменения пароля), поэтому знание

хэш-представления ничего не дает злоумышленнику, который получил доступ к базе данных.

Существует несколько алгоритмов хэширования. Самым простым алгоритмом хэширования является алгоритм MD5. Для хэширования требуется создать специальный провайдер (объект типа MD5CryptoServiceProvider), затем хэшировать набор байтов сообщения. В результате хэширования получается другой набор байт, который в целях хранения можно преобразовать в символьную строку.

```
// читаем сообщение с клавиатуры
string data= ". . .";
// получаем массив байтов сообщения
byte [] bdata=Encoding.UTF8.GetBytes(data);
// создание провайдера для хэширования
MD5CryptoServiceProvider md5Provider =
                                  new MD5CryptoServiceProvider();
// получение хэшированного массива байтов
byte[] hashResult = md5Provider.ComputeHash(bdata);
// получение строки из хэшированных байтов
data = Encoding.UTF8.GetString(hashResult);
      // дальнейшая работа с хэшированной строкой
     .     .     .
```

Среди механизмов шифрования информации, выделяются симметричное и асимметричное шифрование. ***Симметричное шифрование*** – это способ, при котором шифрование и расшифровка производится одним и тем же ключом. В отличие от него, ***асимметричное шифрование*** предполагает, что используются два ключа – открытый и закрытый. Открытый ключ широко распространяется между пользователями и предназначен для шифрования информации. Он не может расшифровать сообщение, которое было им зашифровано. Закрытым ключом владеет только та сторона, которой требуется расшифровка информации. Только он может расшифровать то, что зашифровал открытый ключ.

В зависимости от алгоритма шифрования используются различные провайдеры, однако основная схема работы с ними аналогична. Так, создание объекта-провайдера с помощью конструктора приводит к генерации ключей для

шифрования. Если нужно их сохранить или, напротив, изменить на известные, нужно использовать специальные методы, которые позволяют сохранить или загрузить представления параметров ключа в виде xml-формата (ToXmlString(), FromXmlString()). После настройки ключей шифрование информации осуществляется методом Encrypt(), в расшифровка – методом Decrypt().

Для проверки целостности используются те же средства криптографии, т.е. шифрование и хэширование. Одним из механизмов проверки целостности является понятие ***цифровой подписи документа***. Что такое цифровая подпись? По сути, цифровая подпись представляет собой зашифрованное хэш-значение основных данных документа. Дополнительное шифрование хэш-значения с помощью ключа повышает надежность проверки целостности документа. Итак, когда документ сформирован или легальным способом отредактирован, производится генерация цифровой подписи. Далее подпись хранится вместе с данными документа или отдельно от него. Пусть каким-то способом документ был изменен. Если это произошло нелегальным образом, цифровая подпись не будет сгенерирована по новым данным. Следовательно, повторив процесс генерации цифровой подписи и сравнив сохраненный и новый сгенерированный код, мы узнаем о несовпадениях и тем самым выявим нарушение целостности документа. Так, например, изменения в базу данных или другой источник данных безопасно будет осуществлять только после того, как будет пройдена проверка целостности данных документа.

Приведем типовой код генерации цифровой подписи для нашего учебного приложения, а также типовой код проверки целостности документа-чека.

Добавим на форму две кнопки: «Сгенерировать цифровую подпись» и «Верифицировать документ», которые должны осуществлять действия с выбранным пользователем чеком. Обработчики нажатия этих кнопок приведены ниже.

При генерации цифровой подписи создадим объект чека и сериализуем его в оперативную память. Сериализованный объект представляем как массив байтов, после чего шифруем его с помощью соответствующего провайдера. Получа-

ем из объекта-провайдера ключ и сохраняем параметры подписи – ключ и саму цифровую подпись в файл.

При верификации документа будут проделаны все действия по получению байтового преставления сериализованного объекта. После считывания цифровой подписи из файла с помощью метода VerifyData() соответствующего объекта-провайдера производится сравнение цифровой подписи и набора байтов данных, на основании которого делается вывод о результате верификации.

```
// обработчик нажатия кнопки "Сгенерировать цифровую подпись"
private void gen_Click(object sender, EventArgs e)
{
      // создаем объект выбранного чека
      // как и в примере из предыдущего раздела
      // В результате имеем заполненный
      // объект класса Check с именем chek
      if (dataGridView1.SelectedRows.Count == 0)
           return;
      //получение номера текущего выбранного чека
      int nom = (int)dataGridView1.SelectedRows[0].
                                        Cells["НомерЧека"].Value;
      //получение даты текущего выбранного чека
      DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                         Cells["ДатаЧека"].Value;
      // инкапсуляция данных для показа в печатной форме
      Check chek = CreateCheck(nom, date);

      // подпись будет создаваться для сериализованного
      // объекта класса Check
      // проводим SOAP сериализацию объекта chek
      // создаем объект-форматер
      SoapFormatter ser = new SoapFormatter();

      // создаем поток для сериализации объекта
      // в оперативной памяти
      MemoryStream ms = new MemoryStream();
      // проводим сериализацию в память
      ser.Serialize(ms, chek);

      // считываем сериализованные данные – в результате
      // получаем массив байт
      // переходим в начало потока в памяти
      ms.Seek(0, SeekOrigin.Begin);
```

```
        // получаем массив байт, определяющий объект чека,
        // считывая из потока в памяти
        byte[] message = new byte[ms.Length];
        ms.Read(message, 0, (int)ms.Length);

        // получаем цифровую подпись с помощью алгоритма DSA
        DSACryptoServiceProvider dsa =
                                  new DSACryptoServiceProvider();
        // цифровая подпись – это также набор байт.
        byte[] signature = dsa.SignData(message);
        // сохраняем параметры ключа в виде структурированной строки
        string key = dsa.ToXmlString(true);
        // сохраним подпись данного документа в бинарном файле,
        // имя которого зависит от номера чека
        BinaryWriter br = new BinaryWriter(new FileStream("check" +
                      chek.НомерЧека + ".dat", FileMode.Create));
        // сохраняем ключ в созданный файл
        br.Write(key);
        // сохраняем в файл цифровую подпись
        // сначала количество байт
        br.Write(signature.Length);
        // затем саму подпись
        br.Write(signature);
        // закрываем файл с подписью
        br.Close();
        MessageBox.Show("Создана цифровая подпись");
  }

// обработчик нажатия кнопки "Верифицировать документ"
private void verify_Click(object sender, EventArgs e)
{
      // создаем объект выбранного чека
      // как и в примере из предыдущего раздела
      if (dataGridView1.SelectedRows.Count == 0)
            return;
      //получение номера текущего выбранного чека
      int nom = (int)dataGridView1.SelectedRows[0].
                                      Cells["НомерЧека"].Value;
      //получение даты текущего выбранного чека
      DateTime date = (DateTime)dataGridView1.SelectedRows[0].
                                      Cells["ДатаЧека"].Value;
      // инкапсуляция данных для показа в печатной форме
      Check chek = CreateCheck(nom, date);
      // проводим генерацию хэш-значения для объекта chek
      // SOAP сериализация объекта для формирования
```

```
        // цифровой подписи
        SoapFormatter ser = new SoapFormatter();
        // создаем поток для сериализации объекта
        // в оперативной памяти
        MemoryStream ms = new MemoryStream();
        ser.Serialize(ms, chek);
        // переходим в начало потока в памяти
        ms.Seek(0, SeekOrigin.Begin);
        // получаем массив байт, определяющий объект чека
        byte[] message = new byte[ms.Length];
        ms.Read(message, 0, (int)ms.Length);
        // работа с цифровой подписью - считываем подпись и
        // ключ из файла и осуществляем верификацию
        // читаем данные из файла-подписи
        BinaryReader br = new BinaryReader
                  (new FileStream("check" + nom + ".dat",
                   FileMode.Open));
        // читаем ключ для шифрования
        string key = br.ReadString();
        // читаем данные подписи
        int n_sign = br.ReadInt32();
        byte[] b_sign = br.ReadBytes(n_sign);
        br.Close();
        // импортируем параметры в провайдер шифрования
        DSACryptoServiceProvider dsa =
                                  new DSACryptoServiceProvider();
        dsa.FromXmlString(key);
        // проводим верификацию подписей
        // первый параметр - данные для проверки
        // второй параметр - цифровая подпись
        if (dsa.VerifyData(message,b_sign))
              MessageBox.Show("Верификация чека пройдена");
        else
              MessageBox.Show("Верификация чека не пройдена");
}
```

Лабораторная работа №4

Внести в приложение, созданное на этапе 3, возможности по проверке целостности документов. Генерация цифровой подписи и верификация должны осуществляться не по команде пользователя, а должны быть внедрены в основные потоки команд работы с документами. Придумать и продемонстрировать ситуацию несанкционированного внесения изменений в документ и способ обнаружения этой ситуации.

Советы и типовые ошибки

- При записи и последующем чтении сериализованного объекта из потока, связанного с оперативной памятью, не забудьте перевести курсор на начало потока перед считыванием информации.
- Не забудьте сохранять ключ для генерации подписи и загрузку этого ключа в провайдер.
- Важно, что цифровая подпись получается из набора байт, который можно получить разными способами. При верификации набор байт должен быть получен тем же способом.

СПИСОК ЛИТЕРАТУРЫ

1. Куняев, Н. Н. Документоведение [Электронный ресурс]: учебник / Н. Н. Куняев, Д. Н. Уралов, А. Г. Фабричнов; под ред. проф. Н. Н. Куняева. – М.: Логос, 2012. – 352 с. URL: http://znanium.com/bookread2.php?book=469013.

2. Кабашов, С.Ю. Электронное правительство. Электронный документооборот. Термины и определения [Электронный ресурс]: Учебное пособие / С.Ю. Кабашов.–М.: НИЦ ИНФРА-М, 2013. – 320 с. URL: http://znanium.com/bookread.php?book=410730.

3. Куняев, Н. Н. Конфиденциальное делопроизводство и защищенный электронный документооборот [Электронный ресурс]: учебник / Н. Н. Куняев, А. С. Дёмушкин, А. Г. Фабричнов; под общ. ред. Н. Н. Куняева. – М.: Логос, 2011. – 452 с. URL: http://znanium.com/bookread2.php?book=468998.

4. Александрова, Л.А. Документоведение [Текст]: Учебное пособие / Л.А. Александрова – Казань: Изд-во Казан. гос. техн. ун-та, 2010. – 196 с.

5. Советов, Б. Я. Базы данных: теория и практика [Текст]: учебник для бакалавров: для студентов вузов, обучающихся по направлениям "Информатика и вычислительная техника" и "Информационные системы" / Б. Я. Советов, В. В. Цехановский, В. Д. Чертовской. – Издание 2-е. – Москва: Юрайт, 2012. – 463 с.

6. Андрианова, А. А. Объектно-ориентированное программирование на C# [Текст]: Учебное пособие / А. А. Андрианова, Л. Н. Исмагилов, Т. М. Мухтарова . – Казань: Казанский (Приволжский) федеральный университет, 2012. – 140 с.

7. Шилдт, Г. C# 4.0: полное руководство [Текст]: пер. с англ./ Герберт Шилдт. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.

8. Троелсен, Э. Язык программирования C# и платформа .NET 4 [Текст]: пер.с англ. / Эндрю Троелсен. – М.: ООО ООО «И.Д. Вильямс», 2011. – 1392 с.

9. Дейтел, Х. C# в подлиннике. Наиболее полное руководство [Текст]: пер.с англ. / Харви Дейтел, Пол Дейтел. – СПб: БХВ-Петербург, 2006 г. – 1056 с.

Учебно-методическое пособие

Андрианова Анастасия Александровна
Лаврентьева Елена Евгеньевна
Рубцова Рамиля Гакилевна

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ДЛЯ ПРОГРАММИСТОВ ПО КУРСУ «ДОКУМЕНТОВЕДЕНИЕ»

Подписано в печать 02.03.2016.
Бумага офсетная. Печать цифровая.
Формат 60x84 1/16. Гарнитура «Times New Roman». Усл. печ. л. .
Тираж экз. Заказ

Отпечатано с готового оригинал-макета
в типографии Издательства Казанского университета

420008, г. Казань, ул. Профессора Нужина, 1/37
тел. (843) 233-73-59, 233-73-28